CRYPTANALYSIS OF PSEUDORANDOM NUMBER GENERATORS IN WIRELESS SENSOR NETWORKS

THESIS

Kevin M. Finnigin, 1st Lt, USAF

AFIT/GIA/ENG/06-05

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U.S. Government.

AFIT/GIA/ENG/06-05

CRYPTANALYSIS OF PSEUDORANDOM NUMBER GENERATORS IN WIRELESS SENSOR NETWORKS

THESIS

Presented to the Faculty

Department of Computer and Electrical Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science

Kevin M. Finnigin, BS

1st Lt, USAF

March 2006

AFIT/GIA/ENG/06-05

CRYPTANALYSIS OF PSEUDORANDOM NUMBER GENERATORS IN WIRELESS SENSOR NETWORKS

Kevin M. Finnigin, BS

1st Lt, USAF

Approved:

/SIGNED/	3/6/2006
Barry E. Mullins, Ph.D., AFIT (Chairman)	Date
/SIGNED/	3/6/2006
Henry B. Potoczny, Ph.D., AFIT (Member)	Date
/SIGNED/	3/6/2006
Richard A. Raines, Ph.D., AFIT (Member)	Date

To Theresa,

for her love and support

Acknowledgments

I would like to express my sincere appreciation to my faculty advisor, Dr. Barry Mullins, for his guidance and support throughout the course of this thesis effort. The insight and experience was certainly appreciated. I would, also, like to thank my sponsor, Mr. Neal Ziring, from the National Security Agency for both the support and latitude provided to me in this endeavor.

Kevin M. Finnigin

Table of Contents

Page

Acknowledgments....v
Table of Contents.... vi
List of Figures....x
List of Tables xi
Abstract.... xii
I. Introduction 1
1.1 Objectives....2
1.2 Implications....2
1.3 Preview....3
II. Background 4
2.1 Introduction....4
2.2 Distributed Sensor Networks 4
2.2.1 Sensor Network Components.... 5
2.2.1.1 Mote Classes.... *5*
2.2.1.2 Resource Constraints.... *8*
2.2.2 Popular Hardware and Software Platforms.... 8
2.2.2.1 Mica2.... *9*
2.2.2.2 TinyOS.... *10*
2.3 Security in Distributed Sensor Networks....11
2.3.1 Attacks and Vulnerabilities.... 11
2.3.2 Key Management 12
2.3.2.1 Shared Key Distribution Schemes *13*

2.3.2.2 Public Key Distribution Schemes *15*

2.3.3 Security Protocols in Distributed Sensor Networks 16

2.3.3.1 SPINS: Security Protocols for Sensor Networks *16*

2.3.3.2 TinySec *20*

2.3.3.3 Elliptic Curve Cryptography and EccM 2.0 ..... *21*

2.4 Cryptanalysis 23

2.5 Random Number Generation 24

2.5.1 Linear Feedback Shift Registers 25

2.5.2 Linear Congruential Generators 27

2.5.3 Cryptographically Secure PRNGs 27

2.6 Summary 28

III. Methodology 29

3.1 Problem Definition 29

3.1.1 Goals and Hypothesis 29

3.1.2 Approach 31

3.1.3 Assumptions and Limitations 32

3.2 System Boundaries 33

3.2.1 The Encryption Breaking System 34

3.2.2 The PRNG Performance System 35

3.3 System Services 35

3.4 Workload 37

3.5 Performance Metrics 38

3.6 Parameters 38
3.6.1 The EBS System and Workload Parameters 39
3.6.2 The PPS System and Workload Parameters 40
3.7 Factors 40
3.8 Evaluation Technique 42
3.9 Experimental Design 43
3.10 Analysis and Interpretation of Results 44
3.11 Summary 44
IV. Analysis and Results 46
4.1 Encryption Breaking System 46
4.1.1 RandomLFSR Analysis 46
4.1.2 EccM 2.0 Analysis 48
4.1.3 Encryption Breaking System 51
4.1.3.1 The Expected Performance of EBS *52*
4.1.3.2 Results and Analysis of EBS Performance *53*
4.2 PRNG Performance System 60
4.2.1 A Maximal Linear Feedback Shift Register 60
4.2.2 TinyOS 2.0's Multiplicative Linear Congruential Generator 64
4.2.3 Performance of Alternatives 65
4.3 Summary 66
V. Conclusions and Recommendations 68
5.1 Restatement of the Problem and Conclusions 68
5.2 Contributions and Significance 69

5.3 Recommendations for Future Research 69

5.4 Summary 71

Appendix A: Derivation of Equation Two.....72

Appendix B: Computing the Average Private-Public Key Operation 73

Appendix C: Data Tables.....75

Appendix E: Optimization of Assembly.....78

Appendix F: EBS Java Code 80

Appendix F: MaximalLFSR Code.....115

Bibliography 117

List of Figures

Figure Page

1. A Hierarchical Structure for Wireless Sensor Network Components 6

2. A Time-Release Key Chain for Source Authentication [PSW01].....20

3. Diagram of an 8-bit Linear Feedback Shift Register.....26

4. The Encryption Breaking System.....34

5. The PRNG Performance System 36

6. A Typical Distribution of the Numbers Produced by RandomLFSR.....49

7. EccM 2.0 Algorithm for Generating the Private Key.....50

8. Average Times for Loading Key File for Each Mote.....54

9. The Correlation of RandomLFSR Sequence Lengths and Size of the Key Space Assuming Motes are Addressed Sequentially.....56

10. The Measured Time to Find Private Keys Versus Expected Time.....57

11. The MaximalLFSR Function for Generating the Next Random Number 62

12. Output of Program for Calculating Statistics of Time to Compute Private-Public Key Operations.....73

13. Assembly Code for MaximalLFSR 79

List of Tables

Table Page

1. Key Size Equivalencies for Desired Bits of Security 22

2. The Parameters for the SUTs Categorized by System and Workload 39

3. Factors and the Levels for the Systems Under Test.....40

4. The Various Sequence Lengths Produced by TinyOS 1.1.0's RandomLFSR.....47

5. Sequence Length of First 27 Sequential Mote IDs 52

6. Expected Time to Identify a Private Key Versus Average File Load Time 54

7. Measured Time of the EBS to Identify Private Keys with Zero Rekeys 55

8. The Average Time for the EBS to Identify All Keys in a Network of a Given Size Assuming an Arbitrary Number of Rekeys 56

9. The Expected Time of the EBS to Find a Private Key Given a Mote Producing the Listed Sequence Length.....59

10. A Static Analysis of Alternative PRNGs Versus RandomLFSR.....66

11. Statistics for Average File Load Times 75

12. Statistics For Network Size = 3 76

13. Statistics for Network Size = 6 76

14. Statistics for Network Size = 9 76

15. Statistics for Network Size = 27 77

16. Expected Performance for Sequential Network Sizes 77

AFIT/GIA/ENG/06-05

Abstract

This work presents a brute-force attack on an elliptic curve cryptosystem implemented on UC Berkley's TinyOS operating system for wireless sensor networks. The attack exploits the short period of the pseudorandom number generator (PRNG) used by the cryptosystem to generate private keys. The attack assumes a laptop is listening promiscuously to network traffic for key messages and requires only the sensor node's public key and network address to discover the private key. Experimental results show that roughly 50% of the address space leads to a private key compromise in 25 minutes on average. Furthermore, approximately 32% of the address space leads to a compromise in 17 minutes on average, 11% in 6 minutes, and the remaining 7% in 2 minutes or less. Two alternatives to the PRNG are examined that mitigate the brute-force attack. The alternatives are implemented on the Mica2 mote and examined to determine CPU cycles for execution and memory requirements. The recommended PRNG requires 73 CPU cycles in the worst case and uses 66 bytes of memory. The period of the PRNG is uniform for all mote addresses and theoretically requires 6.6 years on average for a key compromise for the attack used in this thesis.

CRYPTANALYSIS OF PSEUDORANDOM NUMBER GENERATORS IN WIRELESS SENSOR NETWORKS

I. Introduction

Wireless Sensor Networks (WSNs) hold the potential to revolutionize the fields of remote automation and sensing. Applications range from non-invasive habitat monitoring to battlefield surveillance. For sensor networks to reach their full potential, security must be a consideration in designing the hardware and software for future applications. Without security, WSNs could be rendered useless with simple denial-of-service attacks or covertly monitored to learn confidential information. In a worst-case scenario, the devices themselves could be subverted and used to distribute false information to the listener. In data gathering applications such as habitat monitoring, the data is simply lost or misinterpreted. In more sensitive applications, such as battlefield surveillance, vulnerabilities could lead to loss of human life.

Sensor Network Security (SNS) is a growing field of research that presents researchers with challenging goals under tight design constraints. The need for creative and innovative security protocols is clear since current security primitives are too resource intensive in terms of the power, memory, and processing capabilities of sensor network nodes, also known as “motes”.

1.1 Objectives

This thesis focuses on one piece of the overall security picture for WSNs, namely the topic of pseudorandom number generators (PRNGs) as they apply to SNS protocols. Although a small, often overlooked piece of security algorithms, PRNGs hold the potential to open up serious vulnerabilities in security algorithms. This thesis demonstrates the weakness of a particular PRNG and how it leads to a brute-force attack on a state-of-the-art cryptographic protocol. SNS protocols must be based on a strong, sound foundation for future research and applications to move forward. In an effort towards that end, this thesis also examines alternatives to the current PRNG and analyzes the cost of implementing these algorithms.

1.2 Implications

This research attempts to provide a strong foundation on which to build future SNS protocols. In addition, a thorough analysis of the de facto PRNG in use on the widely distributed TinyOS operating system expects to yield valuable information to engineers and scientists that design WSNs. The PRNG of TinyOS 1.1.x contains several modifications to a well understood PRNG design [Sch96]. These modifications are not documented in the code and the reasons for their introduction are not supported by any of the known methods for altering the PRNG design [Sch96]. However, the PRNG continues to be supported as is evidenced by its distribution with the beta release of TinyOS 2.0. This research attempts to shed some light on the PRNG of TinyOS 1.1.x and spread the word on its deficiencies. Ideally this results in engineers spending less

time tuning their application to compensate for the PRNG and spending more time developing the application.

1.3 Preview

Chapter 2 introduces the reader to important background information relevant to information discussed in subsequent chapters. The reader is also briefly introduced to related areas of SNS research. Chapter 3 states the problem and discusses the methodology used to solve this problem. Chapter 4 presents the results and analyses the data. Finally, Chapter 5 concludes by restating the problem, discussing the contributions and significance of the findings and identifying areas for future work.

II. Background

2.1 Introduction

Sensor Network Security (SNS) is a growing field of research that presents researchers with challenging goals under tight design constraints. The need for creative and innovative security protocols is clear since standard security primitives are too resource intensive in terms of the power, memory, and processing capabilities of sensor network nodes, also known as "motes". This chapter introduces SNS and the problem of key management in this design space.

Section 2.2 introduces distributed sensor networks (DSN) and current technologies related to DSNs. Section 2.3 discusses the sensor network security and introduces several problems related to security in sensor networks specifically focusing on the problem of key management. Section 2.4 introduces cryptanalysis and describes several variations of a cryptanalytic attack. Finally, Section 2.5 presents random number generators and examines the pseudorandom number generator (PRNG) used in DSN software.

2.2 Distributed Sensor Networks

DSNs are a sub-class of ad hoc networks. The key differentiation between ad hoc networks and traditional networks is the absence of network infrastructure. Generally speaking, ad hoc networks operate wirelessly, support distributed routing and support some level of self-organization. Ad hoc networks can be broken into subclasses such as Mobile Ad hoc Networks (MANETs) in which most or all of the nodes are mobile or

distributed sensor networks in which most or all of the nodes possess a sensing capability.

This section introduces the concept of distributed sensor networks. First, the reader is introduced to the basic components of sensor networks. In addition, the demand for low cost components is explained and the impact this has on the design of sensor network technology is examined. Finally, popular hardware and software platforms are introduced.

2.2.1 Sensor Network Components

An individual node in a sensor network is called a mote. Sensor networks are envisioned to contain hundreds to thousands of these motes depending on the specific application. Networks are created in an ad-hoc fashion and, besides the motes themselves, have very little need for infrastructure. The networks are self organizing and, depending on the application, can be distributed randomly or by purposefully placing nodes at predetermined points, e.g., at hallway intersections inside a building [ASS01, TAH02].

2.2.1.1 Mote Classes

Several variations of motes exist today. Most motes maintain some type of sensing capability and wireless communication capacity. However, the actual motes themselves vary to fit a wide range of functionality. DSNs require a tiered architecture that results in a hierarchy of nodes with different capabilities at each level [HHK04]. Figure 1 illustrates this hierarchy. It is referred to throughout the rest of this section. An important aspect of DSNs is that information typically flows up the hierarchy.

Some nodes perform sensing functions and represent specialized nodes of very limited resources. These nodes are akin to UC Berkley's Smart Dust motes [KKP99]. They are the lowest rung in the hierarchy in Figure 1. They typically do not

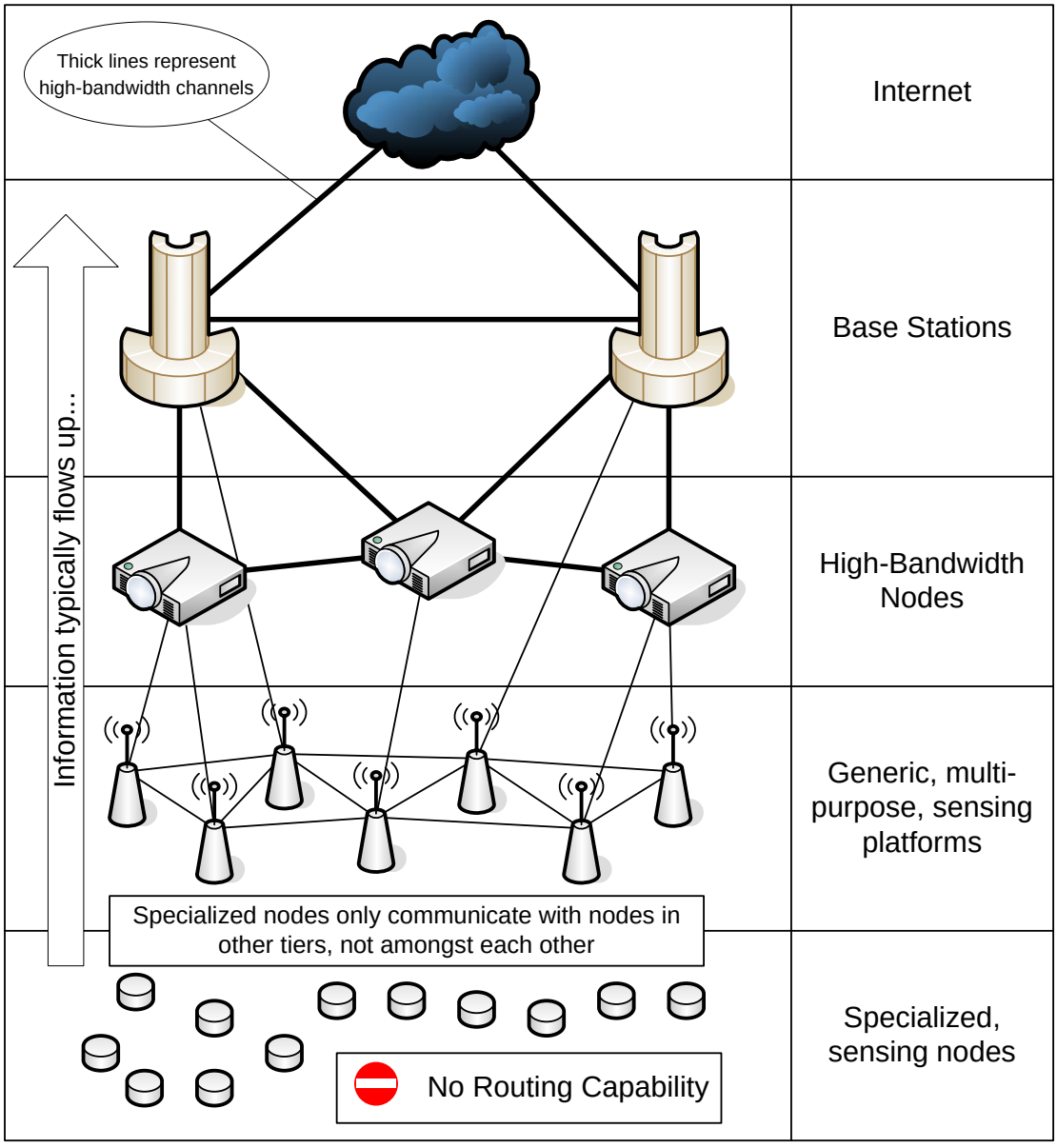

Figure 1. A Hierarchical Structure for Wireless Sensor Network Components

communicate with each other, but instead communicate with nodes higher in the chain either when reporting information or when they are queried for data. A step above these nodes is generic sensors that represent the workhorse of the sensor node classes. These nodes are capable of interfacing with a wide variety of temperature, light, sound, motion, pressure and heat sensors, among others. They have sufficient computation, battery and memory capacity to process, store and transmit sensed and received data to other nodes in the network. A typical example of this mote is the Mica and second-generation Mica2 [HiC02]. These nodes often communicate amongst each other, forming a multi-hop network capable of communicating over greater distances than any single node.

Other nodes are designed for specific sensing functions such as video and audio. These nodes have a greater capacity for computation, battery life and bandwidth then generic sensor nodes. These high-bandwidth nodes also act as superhighways in a DSN reducing the load on less powerful nodes [HHK04]. The application that the DSN is designed for typically determines the number and concentration of these nodes, which are typically fewer in numbers than generic-sensing platforms as shown in Figure 1.

Finally, most discussions of sensor networks involve the concept of a base station. The base station acts as a gateway to more traditional networks, hence this type of node is referred to as a gateway node. These nodes are designed with the state-of-the-art technology and typically cost significantly more than other nodes in the network [HHK04]. Figure 1 shows that more than one of these gateways may be present in a DSN. When using two gateways, it allows the designers to place the nodes at the boundaries of a DSN, thereby reducing the traffic load on adjacent nodes.

2.2.1.2 Resource Constraints

Due to the nature of sensor networks in which hundreds to thousands of nodes could be deployed, there is a demand that individual motes be low cost and unobtrusive. The range of environments that motes could be deployed in requires that they be robust and autonomous. These requirements have an impact on the design of the sensor motes and the resources available to potential software applications. Specifically, sensors must operate on a limited power supply since they are inaccessible . Power must be conserved when receiving, processing and transmitting data. Computation and memory capacity are limited to reduce cost and power. In addition, transmission range is limited to conserve power [CKM00]. There are also practical limitations to transmission range, especially in densely populated networks such as DSNs. For example, an unlimited transmission range for every node in a large network results in increased contention for the shared channel. Shorter transmission ranges decreases contention for the channel and increases throughput.

Other constraints weigh in as well on specific types of applications. Many applications, such as surveillance and environmental monitoring, require real-time feedback. Other applications, such as habitat monitoring or scientific studies, may have less real-time constraints, but could possibly require higher fidelity when reporting data [CKM00].

2.2.2 Popular Hardware and Software Platforms

Early hardware platforms came in a variety of configurations and capabilities [KKP99, PoK00, HiC02, HHK04]. Within the past few years, hardware platforms began

to see second and third generation development. Subsequently the research community moved to the de facto standard hardware platform of the Mica2 mote, which is presented in the next section.

Very little documentation exists in the literature on the early operating systems used during the development of the first generation of motes. TinyOS, introduced in [HSW00], filled the need for system software to manage and operate the device. Traditional embedded operating systems such as VxWorks, WinCE, PalmOS, and QNX are designed for embedded PCs and require resources unavailable on a typical sensor platform. Smaller real time executives such as Creem, pOSEK, and Ariel come much closer to matching the resources available provided by DSN hardware [HSW00]. However, these systems are designed for a different operating environment and tend to be control centric and do not support the efficient handling of hardware interrupts. Since DSNs use radios that generate numerous hardware interrupts, this aspect of the operating system becomes critical to power-efficient operation of the mote. This motivated the development of TinyOS and contributed to the fact that it is the de facto operating system used in DSNs. Section 2.2.3.2 describes the operation of TinyOS in detail.

2.2.2.1 Mica2

The Mica2 is an example of the generic sensing node introduced in Section 2.2.1.1. It is a commercially available mote in use by over 100 research organizations [HHK04]. The Mica2 is a third generation sensing platform initially developed at the University of California, Berkley and then later manufactured by Crossbow Technology, Inc., a sensor systems technology company.

The Mica2 mote operates an 8-bit Atmega128L processor running at approximately 8 MHz on two AA batteries with two modes of operation, active and sleep. It provides 128 kilobytes (KB) of programmable flash memory, 512 KB of EEPROM, and 4 KB internal SRAM. The processor uses 8 milliamperes (mA) and ~100 microamperes (μA) when active and asleep respectively [SHC04].

2.2.2.2 TinyOS

TinyOS is a multi-threaded, event-based operating system designed for sensor motes [HSW00]. It consists of a scheduler and a hierarchy of components. A component consists of command handlers, event handlers, a bundle of tasks, and a frame. The frame in this context is a fixed-size stack frame for storing the state of the currently running program. Commands, events, and tasks execute within the context of a frame. Higher level components issue commands to lower level components, and lower level components signal events to higher level components [HSW00].

A set of tasks provide functionality for calling lower level commands, signaling higher level events, and scheduling other tasks. Tasks are atomic with respect to other tasks and execute asynchronously with respect to events. The obvious benefit of this design is the allocation of only one stack to the currently executing stack, which creates substantial savings in the memory constrained architecture of sensor networks [HSW00].

The atomic nature of tasks allows for the implementation of a simple FIFO task scheduler. When the task queue is empty only an event can result in the scheduling of a new task. This design allows for the power efficient use of the processor since an empty task queue indicates that the processor can enter the sleep state. Hardware peripherals

such as the radio or sensor, can be left active to wake up the processor as needed. These peripherals signal low level event handlers via hardware interrupts, which can trickle up through the component hierarchy, scheduling tasks as required by the application [HSW00].

Each component must identify the resources it provides and the resources it requires. The frame is statically allocated during compilation allowing the memory requirements of the program to be determined before execution and preventing the costs associated with dynamic allocation. Communication between components takes the form of function calls [HSW00].

2.3 Security in Distributed Sensor Networks

There are a number of challenges to the deployment of a secure DSN. Not only must the DSN designer consider the resource constraints of the motes, but the lack of infrastructure inherent in ad-hoc networks has security implications as well. The first examination of security in DSNs, as documented in [CKM00], identifies the critical requirements and constraints in Sensor Network Security (SNS). It recognizes that key management presents significant challenges to the deployment of a secure DSN. DSN deployment in hostile environments also presents a unique challenge to security. Nodes are likely to be left unprotected and exposed to physical access. Mote encasings must provide tamper resistance and protection from physical destruction.

2.3.1 Attacks and Vulnerabilities

Many analyses of attacks and vulnerabilities in DSNs focus on routing protocols [WoS02, HPJ03, KaW03, NSS04]. Wood and Stankovic focus on denial-of-service

(DoS) attacks at all levels in the DSN design space. Hu, et al. introduces the "rushing attack" that pertains to on-demand ad hoc routing protocols such as DSR [Joh94] and AODV [PeR99]. Karlof and Wagner perform an in-depth analysis of routing attacks and countermeasures. They note that many security issues in DSN routing also pertain to ad-hoc network routing. However, the defenses developed for these attacks are not directly applicable to the DSN environment. It is explicitly assumed that defenses using public key cryptography are not applicable because they are too expensive. Although countermeasures and defenses are developed in spite of this assumption, it remains to be seen if defenses using public key cryptography are applicable to DSNs in light of work done in [MWS04]. Newsome et al. provide an in-depth analysis of the "Sybil attack" and offer novel defenses against it. A Sybil attack involves a malicious node or nodes behaving as a larger number of nodes in an attempt to subvert network traffic.

2.3.2 Key Management

Key management in DSNs encompasses the problem of establishing a secret, shared key for use in symmetric cryptographic algorithms. Symmetric algorithms use the same key for encryption and decryption. Thus, modern computer networks require a secure channel to distribute shared keys. The only secure channel available in DSNs is the configuration phase when code is uploaded to the mote prior to deployment. [CKM00] discusses in depth the constraints of DSNs as they relate to key management. [HLV04] examines various energy and memory tradeoffs in the context of security and key management.

A novel approach to key management involves the distribution of some *k* number of keys on each mote where *k* is much smaller than the network size [EsG02, CPS03, DCL04, DDH03, LiN03]. These schemes rely on pre-distribution of key material before deployment to ensure trust when deployed. These schemes are touched on briefly in the next section.

2.3.2.1 Shared Key Distribution Schemes

One way to establish trust is to configure each node with a single, shared master-key during pre-deployment. This key management scheme is easy to deploy and maintain, however a compromise of a single node results in a compromise of the entire network. Alternatively, each node is configured with *N-1* keys, where *N* is the number of nodes in the network. This approach has the nice property of a compromise only affecting *N-1* links in the network. However, it is inefficient since it is unlikely that all nodes can talk to each other and adding nodes after deployment requires installing keys on nodes that may be inaccessible.

The third scheme relies on pre-distributing a set of keys, known as a key ring, of cardinality *k* on each node and uses random graph theory to ensure graph connectivity. [EsG02] first proposed this key management scheme for sensor networks. During a set-up phase, an offline process generates a key pool, *P*, consisting of a large number of unique keys. A key ring is created by randomly selecting k keys without replacement from *P*. *N* key rings are created from the same *P*, where *N* is the network size.

Once all the motes are configured with a key ring they begin a shared key discovery phase. The connectivity of the network is determined by three factors, namely

the size of the key pool, the key ring, and the expected number of nodes in a motes neighborhood, i.e. the number of nodes within a motes transmission range. Since it is a random deployment, the size of the neighborhood is determined via graph theory, which uses the area of deployment, transmission range, and network size. For example, a mote with an infinite transmission range has *N-1* neighbors. The memory constraints of the sensor motes determines the size of the key ring. The network size, deployment area and transmission range are chosen by the network designers. Thus, two of the three factors in the scheme proposed by [EsG02] are DSN constraints. However, the size of the key pool is not a DSN constraint and is scaled to meet the security and connectivity needs of the network.

[CPS03] expands on the scheme developed by [EsG02] by proposing a *q*-composite scheme where *q* keys must be shared between nodes in order to establish a link. [DDH03, LiN03] propose two similar, but different, approaches to random pre-distribution schemes than proposed in [CPS03, EsG02]. [LiN03] uses a polynomial-based key distribution protocol developed for group communication [BSH98] in combination with the pre-distribution scheme proposed in [EsG02]. [DDH03] uses a symmetric key generation protocol proposed in [Blo85] combined with [EsG02].

Random key pre-distribution provides a framework for establishing trust without preexisting infrastructure. In this scheme, trust is delegated to the initialization process that takes place before deployment. This implicitly assumes that the initialization process is free from tampering. Also, re-keying, revocation, and adding nodes to the network requires a certain amount of contingency planning when generating the key pool and key

rings. [EsG02] provides anecdotal evidence that the size of the key ring increases at a slower rate than the size of the key pool for a given connectivity constraint. This seems to indicate that key ring storage requirements are not a constraining factor in implementing this key management scheme despite increased overhead.

2.3.2.2 Public Key Distribution Schemes

Public key cryptography provides a way to distribute keys over an insecure channel. The algorithms used in public key cryptography are known as asymmetric since the keys used for encryption and decryption are not the same. [DiH76] first proposed a key exchange protocol for establishing a shared secret over an insecure channel. The protocol, known as Diffie-Hellman, is based on the discrete logarithm problem (DLP). DLP is the problem of calculating k for some $x = g^k$ given knowledge of $x \bmod m$ and g. This problem is presumed difficult and for a large prime m is considered computationally infeasible as there is no known efficient algorithm [Mal04, DiH76].

Public key cryptography provides a memory efficient way to establish shared keys in DSNs over insecure channels. There is an additional cost in computation however. One implementation of Diffie-Hellman on the Mica2 showed decryption operations, as opposed to encryption operations, would take “tens of minutes” [WKC04]. Since it is an asymmetric algorithm, one operation can take significantly longer than the other. For this reason, public key cryptography was deemed infeasible on DSNs [PWS01, EsG02, CPS03, DCL04, DDH03, LiN03].

2.3.3 Security Protocols in Distributed Sensor Networks

The following sections examine three security protocols in detail. These protocols are chosen to provide background and insight into the potential for SNS.

2.3.3.1 SPINS: Security Protocols for Sensor Networks

Some of the earliest work done in sensor network security focused on the feasibility of security on the resource constrained networks [PSW01]. As a proof of concept, SPINS was developed and implemented on UC Berkley's "Smart Dust" nodes using the TinyOS operating system. Although the nodes themselves were quite typical of a resource constrained node in a sensor network, the network itself consisted of a relatively small number of nodes.

The security goals set in [PSW01] included data confidentiality, authentication, integrity, and freshness. Two security building blocks developed by Perrig, et. al. helped accomplish these goals. The Secure Network Encryption Protocol (SNEP) provided two-party data confidentiality, authentication, integrity, and freshness. The micro version of the Timed, Efficient, Streaming, Loss-tolerant Authentication (TESLA) protocol referred to in [PSW01] as μTESLA provided authenticated broadcast.

Among their greatest contribution was SNEP (Secure Network Encryption Protocol). SNEP provided semantic security, data authentication, replay protection, and weak freshness. These are all important security properties. More importantly, SNEP introduced very little additional overhead in communication cost. Transmission in sensor networks happens to be the most expensive operation in the context of power consumption [HSW00, SHC04].

Since the writing of the SPINS paper there has been a concerted effort to add security to deployed sensor networks. However, one of the greatest weaknesses of the SPINS paper was the key distribution scheme. It requires that each node maintain a single, shared master-key. In [PSW04], one of the authors recognized that a secure, efficient key management scheme for DSNs is critical for SNS. The sharing of a single, shared master key is undesirable since it violates two security principles, namely the principles of least privilege and least common mechanism. The principle of least privilege states that "a subject should only be given the privileges it needs in order to complete its task" [Bis03:343]. If SPINS adhered to this principle, a node would only maintain shared-keys for active communication links with its neighbors. There are practical limitations in sensor networks to consider when trying to adhere to this principle. The establishment of a secret key between two nodes in a network incurs a certain amount of overhead in computation and the number of transmissions. However, the principle of least common mechanism is more feasible. It states that "mechanisms used to access resources should not be shared" [Bis03:348]. In SPINS a master key is shared by all nodes to derive shared keys for node-to-node communication. If a single node is compromised then the entire network becomes vulnerable to eavesdropping. As discussed in Section 2.3.2, this is undesirable and ultimately avoidable.

An important factor when considering encrypting messages in DSNs is how the encryption algorithms change the size of the original message. The implementation of SNEP consists of the RC5 block cipher in counter (CTR) mode. Since the CTR mode is used, the ciphertext has the same length as the plaintext. In addition, the same function

can be used for encryption and decryption. Both of these properties are desirable in sensor networks because they save on memory in terms of code space and message length. Since the ciphertext is the same length as the plaintext there is no additional cost in transmission time. Another benefit that the CTR mode provides is weak data freshness. The counter is incremented after each message the sender transmits and therefore the receiver can verify that the received packets have a monotonically increasing counter.

RC5 is subject to a differential cryptanalysis from a chosen plaintext attack [KaY98], but this attack requires approximately 2^{44} plaintext-ciphertext pairs for a successful attack. It seems unlikely that even a large sensor network will generate this much traffic. For example, a network consisting of 10,000 nodes with each node generating an average of 51 encrypted messages per hour operating continuously for a year manages less than 2^{33} messages. A well-publicized application for habitat monitoring [MPS02, SPM04, SOP04] used a sampling rate of once every 70 seconds, which is approximately 51 samples an hour. The above calculation assumes errorless communication, doesn't factor the computation time for encryption, and assumes a mote can actually operate for a year. Thus, 2^{33} messages is an upper bound, or in the context of an attack a best-case scenario, and RC5 appears to be a good choice for SNS protocols.

The implementation of SNEP also provided message authentication via a message authentication code (MAC) that used the same block cipher as the encryption algorithm. For each packet sent over the channel a MAC is computed that not only provided authentication, but data integrity as well. Strong freshness is achieved through the

generation of a random nonce that is sent out with a request message. The reply message includes the nonce in the MAC computation, which allows the node to know that the response was generated in reply to its request.

The authors of [PSW01] saw the need for authenticated broadcast to support node-to-node key agreement and secure, authenticated routing. To that end, they introduced μTESLA, the micro version of the Timed, Efficient, Streaming, Loss-tolerant Authentication (TESLA) protocol. The TESLA protocol [PTS00] is not specifically designed for sensor networks and has a number of shortcomings in this context. However, μTESLA was designed to overcome these shortcomings by reducing the requirements of TESLA. It removes digital signatures from initial packets and instead relies on symmetric mechanisms and a shared master key. Key disclosures in each packet are also removed and the number of authenticated senders is restricted thereby reducing the key storage requirements for each mote.

μTESLA uses the concept of a key chain to provide message authentication. The protocol generates the key chain by randomly choosing the last key in the chain, K_n, that is then supplied to a one-way hash function, F, to generate K_1 through K_{n-1} such that $K_i = F(K_{i+1})$. The keys of the key chain are used in a sequential fashion to create a MAC for a message. This key is later released to provide authentication to nodes that receive the message. μTESLA requires that each receiver must have one authentic key of the one-way key chain, be loosely time synchronized to the sender, and know the key disclosure schedule of the sender. In this way a node can authenticate messages upon addition to the network [PSW01].

Figure 2 illustrates an example of μTESLA. Each hash on the timeline represents one time interval. At each interval, the sender uses a new key to generate a MAC for each packet that is broadcast. The packets are represented by P1, P2, etc. The receiver is assumed to know K_0 through some authenticated channel. The key release schedule is arbitrary, but assume it is two time intervals for this example. Packets P1 and P2 are broadcast with a MAC using K_1. Packet P3 is broadcast with a MAC using K2. At this point, the receiver cannot authenticate any packets. At time interval four, the sender broadcasts K_1, which for illustrative purposes is lost. In the fifth interval, K_2 is broadcast, which can be verified by checking $K_0 = F(F(K_2))$. Thus, P3 can be authenticated. In addition, the receiver knows $K_1 = F(K_2)$ [PWS01].

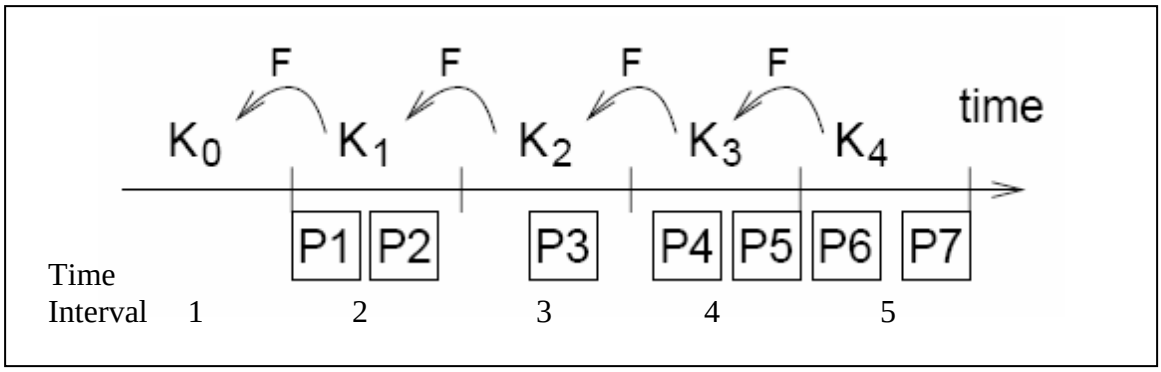

Figure 2. A Time-Release Key Chain for Source Authentication [PSW01]

2.3.3.2 TinySec

TinySec is a fully implemented link layer security architecture for wireless sensor networks [KSW04]. It could be considered the second generation security architecture for wireless sensor networks after SPINS. It is incorporated in the current version of

TinyOS and provides security, performance and ease of use in sensor network applications. The security services of TinySec provide authentication and encryption (TinySec-AE) and authentication only (TinySec-Auth).

Encryption is provided with the RC5 or Skipjack block ciphers using cipher block chaining (CBC) mode. CBC mode typically requires an initialization vector (IV) that is combined with the plaintext to create a pseudorandom bit sequence. For this to work correctly, the IV must not be reused, which means it must be long enough to guarantee that it not be reused. The authors of [KSW04] chose to accept the risk of IV reuse and compensate for this by choosing block ciphers that are robust in the presence of repeated IVs. Authentication is provided using a MAC that is computed and verified using cipher block chaining.

2.3.3.3 Elliptic Curve Cryptography and EccM 2.0

Elliptic curve cryptography uses elliptic curves in conjunction with the discrete logarithm problem (DLP) to implement a public key cryptosystem. This system was first described in [Mil86]. Elliptic Curve DLP (ECDLP) is concerned with the problem of finding k given $P=kG$ and G, where P is the public key and G is the base point on the elliptic curve, which is in the public domain. Based on the mathematics of elliptic curves it is assumed difficult to calculate k and is associated with the same set of problems concerning the factorization of large integers.

The main advantage of ECDLP over traditional public key cryptosystems is that they allow for smaller keys sizes for equivalent security. The United States National Institute of Standards and Technology (NIST) outlines this equivalency in [BBB05].

Table 1 describes the key size equivalencies. The left column describes a level of security in which the best known attack is an exhaustive search. Algorithms for factoring numbers and general discrete logarithm attacks determine the equivalency presented in the other columns. For example, to provide 80 bits of security, the RSA public key algorithm must use a key size of 1024 bits. RSA is a public key cryptosystem first described in [RSA78] and is based on the DLP described in [DiH76]. Alternatively, for an implementation of ECDLP, only a key size of 160 bits is required for 80 bits of security. In other words, the current mathematical tools and theories available to solve the similar, yet different mathematical problems of DLP and ECDLP results in significantly smaller key sizes for elliptic curve cryptosystems.

Table 1. Key Size Equivalencies for Desired Bits of Security

Bits of Security	**RSA**	**Elliptic Curves**
80	1024	160
112	2048	224
128	3072	256
256	16360	512

EccM is the module name within TinyOS for an implementation of ECDLP on the Mica2 platform [Mal04, MWS04]. EccM 2.0 provides public key encryption and decryption for DSNs at a reasonable cost in terms of computation and memory. The generation of a public-private key pair requires approximately 34 seconds. With knowledge of a mote's public key, it takes approximately 34 seconds to generate a shared secret. The latest version of EccM 2.0 requires 35,401 bytes of total memory upon

installation. Running stack size reached a maximum of 81 bytes [MWS04]. The key size is 163 bits, thus providing about 80 bits of security according to Table 1.

2.4 Cryptanalysis

Cryptanalysis is the analysis and deciphering of cryptographic messages or systems. Modern cryptography makes the deciphering of cryptographic messages computationally infeasible. A cryptanalyst attacks such a system by looking for weaknesses in the protocols. Examples of potential vulnerabilities include key exchange protocols, software bugs, and human error.

Attacks are categorized according to the type of attack performed against the cryptosystem. Almost all attacks assume that the cryptographic algorithm is known a priori. A *ciphertext only* attack uses knowledge of only the ciphertext to decrypt the message. This type of attack is usually considered the most difficult to perform [Wag03]. A *known plaintext* attack uses knowledge of both the plaintext and ciphertext to gain knowledge of the key. This type of attack is typically used in an exhaustive search for the key. The attacker usually has a plaintext message and the corresponding ciphertext produced by the cryptosystem. The attacker systematically encrypts the plaintext with a key from the key space to produce a ciphertext. The two ciphertexts are compared and if a match is found then the key is discovered. A *chosen plaintext* attack requires the attacker to have access to the cryptosystem and thus provide a plaintext of his choice for enciphering. This type of attack exploits cryptosystems that produce patterns in ciphertext messages from similar plaintext messages. A *chosen ciphertext* attack is the

same as a chosen plaintext attack, except that the attacker can chose to decipher specified ciphertext messages.

All cryptographic functions are susceptible to a brute-force attack. A brute-force attack systematically searches the key space for the appropriate key. This type of attack is synonymous with an exhaustive search of the key space as described in the previous section. The expected time to find the key is given in equation 1, where t is the time to encrypt a plaintext and n is the size of the key space. To thwart this attack,

$$\frac{1}{2}tn \tag{1}$$

cryptosystems use astronomically large key spaces. The large key space makes it computationally infeasible for an attacker to try all the keys in a reasonable amount of time. However, with the advent of the Internet and distributed computing, processing power is proving to be less of a barrier then previously thought. A widely publicized attack on the Data Encryption Standard (DES) cipher [NBS99], a symmetric key cipher with a 56-bit key, took advantage of distributed computing and the Internet to compute 245 billion keys per second [Nel99] and crack the encryption in a little over 22 hours.

2.5 Random Number Generation

Random number generation has a rich history of success followed by failure. Many attempts at designing random number generators end in failure because they do not pass statistical tests of randomness. Indeed randomness is a subjective concept in and of itself. While a sequence of numbers may pass one test for randomness it may not pass

another. Consequently, it is important to understand the requirements of the system using the random numbers and know exactly how they will be used.

For all intents and purposes, a random sequence is defined for a given set of elements as each element having an independent and equal probability of being chosen as the next element in the sequence. This phenomenon is best illustrated with the flip of a coin. Each flip holds the potential of turning up heads or tails and is independent of all previous flips. Due to this definition, computers, as they are now known, will never be able to produce random sequences. Algorithms attempting to produce sequences that have the statistical properties of a random sequence are known as pseudorandom number generators (PRNGs).

PRNGs require a seed that determines the starting point from which the PRNG will begin generating numbers. PRNGs are predictable by definition. Given the current state of the PRNG the next number can be predicted. Although this flies in the face of a truly random sequence, the next number generated by the sequence can be statistically shown to be independent (i.e., unbiased and uncorrelated).

2.5.1 Linear Feedback Shift Registers

Linear Feedback Shift Registers (LFSRs) are a type a PRNG that generates a sequence of independent, uniformly distributed sequence of 1's and 0's, also known as a bit-stream. The shift register consists of a bit sequence, b_n, …, b_0, where *n* is the length of the LFSR and b_n is the most significant bit. Upon request for the next random bit, the shift register is shifted one bit to the right and b_0 is output. The new bit, b_{n+1}, is calculated by XORing certain elements of the shift register with b_0. The elements of the

LFSR used in the XOR operation are called the tap sequence. Figure 3 illustrates the concept of an LFSR. A maximal LFSR is an LFSR that cycles through 2^n-1 internal states before reaching the initial state again. Zero cannot be reached unless it is the initial state, in which case the LFSR produces nothing but zeros. The tap sequence determines whether or not a sequence is maximal and is actually a well understood mathematical problem [Sch96, Sti02]. Thus, a tap sequence can be derived for an arbitrarily chosen *n*.

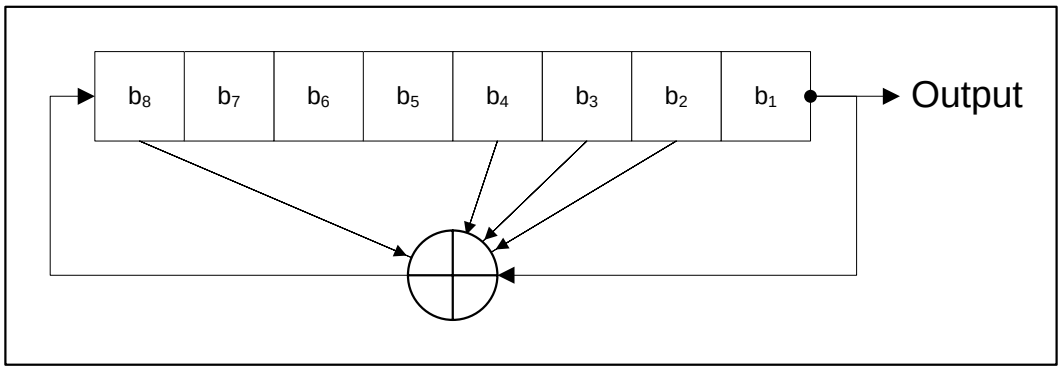

Figure 3. Diagram of an 8-bit Linear Feedback Shift Register

LFSR-PRNGs are best used in cryptography as stream ciphers. LFSR-PRNGs generate a keystream of a length equal to the plaintext from successive calls to the bit-generating function. Each output bit, i.e., the least-significant bit, is used as the next element of the keystream. The plaintext is encrypted using the keystream to get the ciphertext. The encryption operation is usually a simple bit-wise exclusive-or operation.

Due to the popularity and simplicity of LFSR-PRNGs, researchers have thoroughly studied the randomness of the keystream [Sch96]. Several approaches combine two or more LFSRs in an effort to conceal the internal state. Additional LFSRs

complicate the problem, but once the internal state of the LFSR is discovered the next value of the LFSR-PRNG is predictable.

2.5.2 Linear Congruential Generators

Linear congruential generators (LCGs) are a class of pseudorandom number generators taking the form of $s_{i+1} = (as_i + c) \bmod m$ [PaM88, Sch96]. A seed, s_0, is supplied and variables *a*, *c*, and *m* are chosen appropriately. If *a*, *c*, and *m* are chosen with some care, the LCG produces a maximum sequence with a period *m*. A common form of this LCG selects $c = 0$ and is known as a multiplicative linear congruential generator (MLCG). In this form, if *a* and s_0 are relatively prime to *m*, then a maximum sequence is generated. For this reason, *m* is often prime, thus ensuring that *a* and s_0 are relatively prime to *m*.

All that remains for an MLCG to produce a statistically random sequence is the proper choice of *m* and *a*. Countless studies have shown that $m = 2^{31}-1$ and $a = 16807$ withstand both empirical and theoretical tests for randomness [PaM88]. Unfortunately, this PRNG does not provide a cryptographically secure source of random numbers either.

2.5.3 Cryptographically Secure PRNGs

Cryptographically Secure PRNGs (CS-PRNG) aspire to produce random number sequences that withstand cryptanalysis. As the name suggests, this class of PRNGs are suitable for use in key generation algorithms. They provide a level of unpredictability when some or all of its secrets are known.

CS-PRNGs often require certain prerequisites in order to remain secure. For instance, a common prerequisite is that the seed remain secret. This is usually achieved

by creating a seed from a true random source. Random sources in modern computers include packet interarrival times, keyboard latency, and white noise from radio receivers [Sch96]. While all of these sources are adequate, special care must be taken when altering these numbers for use in cryptographic primitives. A level of post-processing to distill any bias or nonrandomness is usually required. Of greatest importance in this discussion of random sources is that the bits cannot be reproduced.

2.6 Summary

This concludes the background discussion. This chapter introduces the terms and concepts necessary for an understanding of the following chapters of this work. Of particular importance to the reader are the sections discussing EccM 2.0 and pseudorandom number generators, Sections 2.3.3.3 and 2.5 respectively. Equation 1 from Section 2.4 is also of importance and referred to frequently throughout the rest of this work.

III. Methodology

3.1 Problem Definition

Given the current state of SNS, it seems likely that a fast, power-efficient, and secure method for generating random numbers in sensor networks is an immediate need. Most key management schemes completely side-step the problem of generating keys on the mote, as is the case in the pre-distribution of secret keys. Others, such as EccM 2.0 use questionable methods for generating keys. The following chapter defines goals and hypotheses related to random number generation in sensor networks and its impact on key generation. It also presents a methodology for achieving these goals.

3.1.1 Goals and Hypothesis

The weakness of the key generation algorithm of EccM 2.0 is known [MWS04]. However, a metric measuring the extent of this weakness is unavailable. EccM 2.0 relies on the underlying operating system, TinyOS, and its PRNG module called RandomLFSR, for generating private keys. RandomLFSR is based on an LFSR design for generating random sequences and is seeded with the mote's ID, which is a 16-bit unsigned integer. The mote ID is also used as the network address and therefore is sent in the packet header. This fact makes the exact random sequence a mote generates discoverable simply by observing network traffic. Since RandomLFSR is based upon a 16-bit algorithm there is a maximum bound on the period of 2^{16}-1. A pilot study performed on RandomLFSR generating all possible sequences shows the period to be about ½ of this maximum in the best case. [MWS04] specifically states EccM 2.0's "reliance on TinyOS's RandomLFSR module is troubling cryptographically". How the reliance on

this PRNG affects the security and strength of the underlying cryptographic primitives of EccM 2.0 is the subject of this thesis. It is hypothesized that the encryption is breakable using a brute-force attack given a mote ID and public key. Specific questions to be answered include:

1. Can the weaknesses of the LFSR-PRNG in TinyOS be exploited to break the public key cryptography of EccM 2.0 and, if so, how fast can it be done?
2. Will the rate of key compromise exceed the motes capacity to rekey?

Another goal is to propose a fast, power-efficient, and secure PRNG that defeats the brute-force attack described in this thesis. Secondary to this goal is to propose a PRNG to replace the current RandomLFSR module. Many of the system functions in TinyOS rely on RandomLFSR to behave in a certain manner. It may even be the case that specific applications are tuned to perform optimally with RandomLFSR's quirky behavior, such as radio interrupt handlers that determine backoff. System performance aside, cryptographic protocols must have numbers that meet requirements of CS-PRNGs regardless of the time to produce them. These requirements for a PRNG on TinyOS represent an apparent dichotomy. RandomLFSR does not allow for this distinction. An objective of the stated goal is to create a parameterized PRNG. This will allow for a PRNG initialized to the same starting conditions as RandomLFSR to perform comparably in terms of execution time, CPU cycles and resource requirements. If the proposed PRNG is implemented, it must not negatively affect the current system functions relying on RandomLFSR. Many candidate PRNGs exist including modifying the current LFSR

design to be a maximal LFSR PRNG. It is not the objective of this work to create a novel PRNG, but instead base the proposed PRNG off a well known PRNG algorithm.

3.1.2 Approach

To break the public key encryption of EccM 2.0, a network of three or more motes randomly sends encrypted messages amongst each other. A malicious mote connected to a laptop promiscuously monitors the network transmissions and relays the captured traffic to a packet analyzer on the laptop. The laptop identifies packet headers and captures the mote IDs of the network. Since the motes use public key encryption, the motes send the keys in the clear, which the malicious mote also captures. Once the laptop identifies a mote's ID and its public key it executes a brute-force attack to find the corresponding private key. This is a modified *known plaintext* attack, except that in this case, the brute-force attack systematically creates a private key and performs the same computation as EccM 2.0 to produce the public key. If the keys match, then the private key is discovered. Theoretically, this attack is possible on any cryptographic system, but is normally infeasible due to the size of the key space that must be tested. EccM 2.0 uses a 163-bit key, but generates these keys with a 16-bit LFSR that has a theoretical maximum period of 2^{16}-1. Thus the key space is significantly reduced and vulnerable to a brute-force attack.

The current implementation of the LFSR-PRNG is analyzed for areas of improvement. The PRNG could be seeded with a random number either captured from the environment or created upon mote configuration. In addition, the selection of taps for the LFSR is examined in order to create a maximal LFSR. Added flexibility is

considered for system processes that require fast random number generation at the cost of a lower quality random number. Alternatively, applications that require high quality random number sequences need to be able to specify this when requesting a random number from the PRNG. Research literature documents feasible alternatives to the LFSR method of generating random numbers that have been proven to be cryptographically secure, but these algorithms may be too resource intensive for sensor network motes.

3.1.3 Assumptions and Limitations

Unfortunately, like many other applications built for TinyOS, EccM 2.0 is still in its infancy. Before EccM 2.0 was developed it was held as common sense that public key systems were too expensive for DSNs. EccM 2.0 was developed to prove that assertion wrong. However, there still remains much to be done. Section 5.3 discusses this in further detail, but suffice it to say EccM 2.0 works as a proof-of-concept. It is meant to work with two motes only and simply broadcasts key messages without any other information. The receiver simply assumes the received key message is from the other mote and begins its calculations for generating the shared secret. Although EccM 2.0 is not fully implemented, it can be reasonably assumed that in a network of three or more motes, a receiver must be able to associate the broadcast of a public key with a mote in the network. It follows that the sender would include its network address within the broadcasted key message. This is critical to the brute-force attack presented in this thesis because it uses knowledge of the mote ID to reduce the key space of EccM 2.0.

Actual measurement of the execution times for the alternatives may be impossible to document when using the system clock of the Mica2 due to its resolution; the smallest

measurable increment is the millisecond. The Mica2 is clocked at 8 million cycles per second. In order to measure any difference in execution times, the PRNGs must differ by roughly 8000 cycles. Unfortunately, at first glance, the RandomLFSR appears too simplistic to require anywhere near this amount of cycles. However, due to its simplicity a static analysis of the number of cycles required to execute the PRNG function may be possible. This provides a much more accurate measurement of the difference between the alternatives.

3.2 System Boundaries

Since Section 3.1.1 identifies two goals, there are two Systems Under Test. The first system is called the Encryption Breaking System (EBS). The second system is called the PRNG Performance System (PPS). In the following discussion of system boundaries and throughout the rest of this thesis, the concept of rekeying is used frequently. In the scope of this thesis, a rekey operation is defined as a private-public key operation in EccM 2.0 in response to a request to communicate with a mote in which a shared key in an active state does not exist. Key states are formally defined in [BBB05], however only the active and deactivated states are necessary for this thesis. The active state of a key is the point in a keys lifecycle in which it may protect information, i.e., perform encryption, and process protected information, i.e., perform decryption. A key in the active state transitions to the deactivated state after a predetermined time period has expired and can only process protected information. The time period that a key in EccM 2.0 remains in an active is defined as the *rekey period*.

3.2.1 The Encryption Breaking System

This system consists of the laptop running the encryption breaking program (EBP). The Component Under Test is the encryption breaking program (EBP). Figure 4 illustrates the system. This SUT addresses the first question stated in Section 3.1. The second question requires a metric for the power cost of generating a private-public pair in EccM 2.0. According to [MWS04] this is approximately 0.00549 Joules for the private key operation and 0.816 Joules for the corresponding public key operation.

It is assumed that a private-public keying operation is only done in response to a request from a neighboring node to communicate securely. This is a safe assumption since it is a waste of time and energy to automatically rekey once the freshness of the current key expires. This assumption implies that with every rekey operation, a shared secret must also be generated. While no energy cost for computing a shared secret is

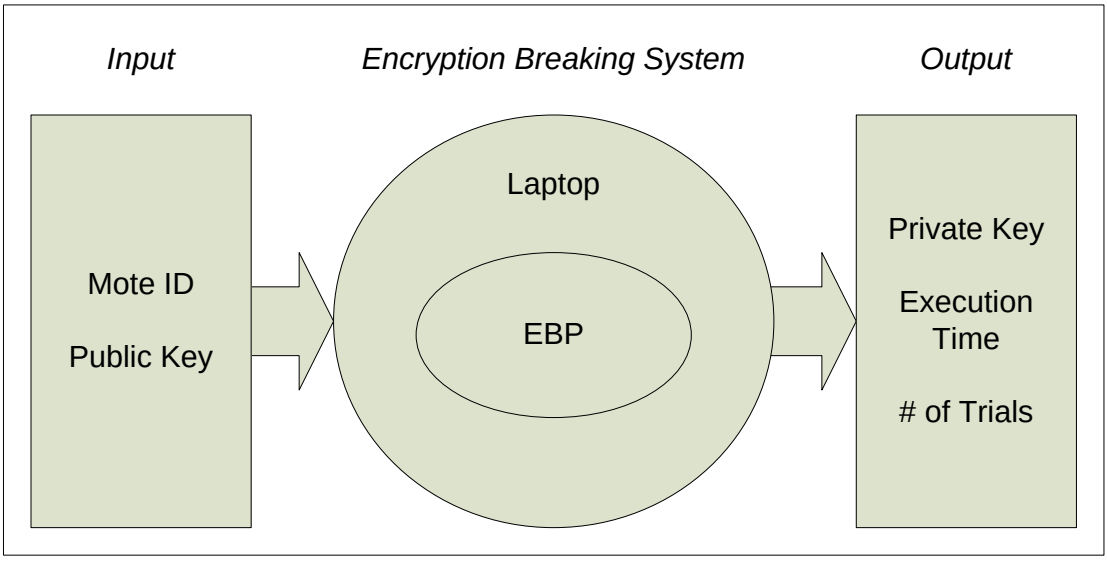

Figure 4. The Encryption Breaking System

supplied in [MWS04] the average time to compute the shared secret is included, which turns out to be 34.173 seconds. Similar times for private and public key operations are provided and they are 0.229 seconds. and 34.161 seconds, respectively. This gives a total running time of 68.563 seconds. If one rekey operation is performed each hour, it would result in a 1.9% duty cycle by the following calculation.

$$\frac{68.563\ \frac{\text{seconds}}{\text{hour}}}{3600\frac{\text{seconds}}{\text{hour}}} \times 100 = 1.904\%$$

3.2.2 The PRNG Performance System

This system consists of a Mica2 mote with the current RandomLFSR module and the proposed PRNGs installed on the OS. The Components Under Test are the proposed PRNGs. This is illustrated in Figure 5.

3.3 System Services

The EBS offers one service, which is a modified *known plaintext* attack on the EccM 2.0 key generation algorithm. The service has two possible outcomes, either success or failure. Success is indicated by the successful match of the given public key with the key generated by the EBP. The primary metrics are the time to find the key, the number of keys tried before a successful decryption, and the private key. Failure occurs if the key space is exhausted without finding a key that successfully matches the given public key. Failure can also occur if the key space is large enough to effectively thwart a brute force attack. While failure in this context is fundamentally subjective, the expected time to compromise a private key and the duty cycle provide a way to formally define

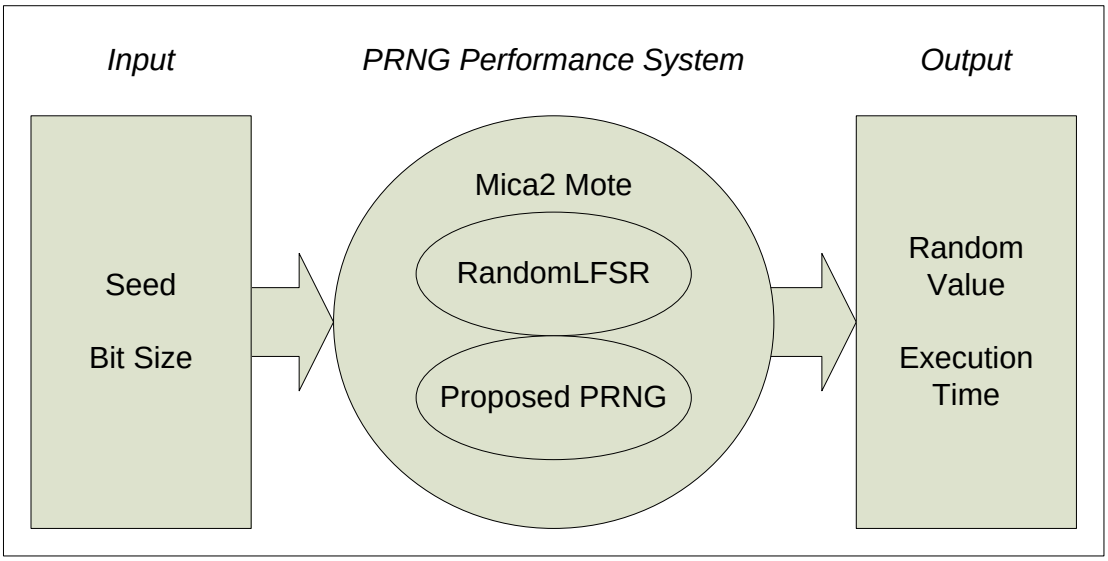

Figure 5. The PRNG Performance System

this concept. Requiring a mote running EccM 2.0 to rekey once an hour requires a 1.9% duty cycle. Requiring the mote to rekey twice an hour requires a 3.8% duty cycle. Alternatively, requiring a rekey once every two hours requires a 0.95% duty cycle. Thus, the expected rate of compromise is directly related to the duty cycle required for a rekey. This is defined in equation 2, where *r* is the rate of private-public key operations per hour on the laptop, n is the size of the key space, and *D* is duty cycle. See Appendix A for the derivation of this equation.

$$\frac{2r}{n} * 1.9 = D \tag{2}$$

Selecting *D* and measuring *r* results in a lower bound for *n*, which precisely defines how large the key space must be to thwart a brute-force attack for a desired duty cycle. Selecting *D* is subjective, but for the purposes of this thesis, a lower bound on the

duty cycle of 0.1% is not unreasonable. Thus, if the size of the key space results in a duty cycle less than 0.1% the system fails.

The PPS offers one service, the generation of random values of a specified number of bits. The service has two possible outcomes, success or failure. Success is determined by a successful generation of a random value of sufficient length. Failure occurs if the PRNG fails to generate a sequence of random values of sufficient length to thwart a brute-force attack, i.e., the PPS fails if the EBS succeeds.

3.4 Workload

The workload for the EBS is directly related to the number of motes in the system. The TinyOS architecture and the network are simulated and the public key is generated and fed to the EBP. Two motes communicating across a channel do not accurately model a computer network because there is no contention for the channel. This means three is the minimum number of motes needed to form a network. Sensor networks are envisioned to have thousands of motes deployed in an ad hoc fashion. However, it is not necessary to scale the workload to this size because an asymptotic upper bound can be found for N, the number of nodes in the network, based on the algorithm used in EBP.

The workload for the PPS is a small application running on a Mica2 mote. The application generates iterative requests to the PRNG modules under test. The application generates requests until the sequence repeats. The period of the PRNG is reported at this time. Practically speaking, it is unnecessary to run the application on a Mica2 mote. The distribution of TinyOS 1.1.x is capable of emulating program behavior on a PC

[LLW03]. The number of CPU cycles are determined from a static analysis of the assembly code and likewise the memory requirements are determined from counting the bytes required for each instruction.

3.5 Performance Metrics

The main performance metrics for the EBS are the amount of time the system takes for a success and the number of keys tried before success. These metrics provide a measure of the feasibility of the EBS. If the EBS takes longer to break the encryption than the motes capacity to generate new keys, the current method for creating random numbers may be considered strong enough. This is unlikely given the size of the key space.

The main performance metrics for the PPS are the time to generate a random value and the memory requirements of the PRNG. The execution time is directly related to CPU cycles on the Mica2 mote and is a sound metric for determining power consumption due to CPU computation. In addition, program memory is at a premium on the Mica2 and should be considered for any application that runs on the Mica2.

Another important factor when considering PRNGs is the randomness of the output and the period of the random sequence. The intent of this thesis is to use published algorithms and verification of these factors is assumed unnecessary.

3.6 Parameters

This section describes the parameters chosen for the two systems under test. Table 2 identifies the system parameters and workload parameters for each system.

Table 2. The Parameters for the SUTs Categorized by System and Workload

	Systems Under Test	
	EBS	**PPS**
System	Laptop CPU speed System Processes	Mote CPU speed Mote memory Mote energy
Workload	Network size Number of rekeys	PRNG period

3.6.1 The EBS System and Workload Parameters

The rate at which the key space is generated and potential keys are tested is directly related to the speed of the CPU on the laptop. Since it is a brute force attack, it is a simple matter of processing power and time. The number of keys to break determines the workload submitted to the system. This is directly related to the network size.

Another workload parameter that affects performance is the number of rekeys a mote performs before an ID-key pair is captured and sent to the EBS. The generation of a private key in EccM 2.0 is an iterative call to the LFSR-PRNG until enough numbers are generated to create a 163-bit key. If an ID-key pair is captured during the initial startup of a sensor network, the EBS only has to generate one key before success. However, if an ID-key pair is captured at an unknown time after deployment it is impossible to know how many calls to the LFSR-PRNG have been made.

Finally, during the pilot study, mote IDs generated random sequences of varying length. The maximum period observed is 31,796, which is about half the theoretical

maximum. The minimum period consists of only one random value. Hence, the seed used for RandomLFSR affects the period of the PRNG and consequently the size of the key space. This behavior of RandomLFSR will be examined in-depth to identify particularly weak seeds (i.e., mote IDs).

3.6.2 The PPS System and Workload Parameters

The execution time of a request for a random value is directly related to the CPU speed of the mote. The number of requests that can be made is ultimately constrained by the mote's power capacity. Finally, if the results need to be saved, the maximum number of random values stored is limited by the mote's memory capacity and the size of the value. The size of the value requested varies according to the workload submitted to the system.

3.7 Factors

This section identifies the factors chosen for experimentation. Table 3 identifies these factors and their respective levels.

Table 3. Factors and the Levels for the Systems Under Test

EBS		**PPS**
network size	3, 6, 9, 27	none
# of rekeys	0, random	

The main purpose of the EBS is to demonstrate that applications that rely on the TinyOS LFSR-PRNG are vulnerable to exploitation; no system parameters identified in the EBS will be varied. The network size and number of rekeys are varied to measure system response. It is standard practice for motes to be assigned IDs starting at 0 and

counting up sequentially during configuration. This same practice is used when assigning mote IDs in the EBS and is not varied in order to reduce the number of experiments. The EBS is expected to perform closely to equation 1, the expected time of a brute-force attack. The key space of a network of motes is simply the summation of each individual mote's key space. To verify this, network sizes of 3, 6, 9 and 27 are selected as levels for experimentation.

The sequence produced by RandomLFSR for a given mote ID can be viewed as a cyclical group. The number of rekeys corresponds to an element of this group. For example, zero rekeys corresponds to the first element of the cycle. An arbitrary number of rekeys, x, corresponds to the *x mod n* element of the group, where *n* is length of the sequence, i.e., the number of elements in the group. To generate a private key in EccM 2.0, 11 calls are made to the LFSR-PRNG to generate enough bits for a 163-bit key. Since the RandomLFSR module is initialized by EccM 2.0, only it may call the LFSR-PRNG, preventing random permutations of the sequence. This could allow calculation of the position in the sequence given the number of rekeys. However, no reasonable way to attain this information seems available. Therefore, simulation of an attacker at network deployment and at an arbitrary time after deployment are only considered. To simulate an attacker at network startup the first key in the sequence is fed to the EBS. To simulate an arbitrary number of rekeys, elements of the sequence are chosen randomly using a uniformly distributed random number generator.

The Mica2 and TinyOS are popular platform choices for sensor networks. Therefore, the CPU speed, memory size, and energy parameters are not considered

variables for this system. The period of the PRNG determines its ability to thwart a brute force attack. However, the period of each alternative is dependent upon its design and cannot be modified.

3.8 Evaluation Technique

The performance evaluation for both systems is done via measurements of real systems. The EBS uses simulation to generate the workload, but the EBP is implemented in Java on a laptop. The laptop contains a 2 GHz AMD Athlon 64 Processor 3200+ with 512 MB RAM. The operating system is Microsoft Windows XP, Home Edition with Service Pack 2 installed. All non-critical system services are stopped and the only running applications are the the Windows Task Manager and a command prompt. The EBS uses the Java Random class as its PRNG for selecting random keys from the key space of each mote. The seed for the simulation is arbitrarily chosen as 1030785277711181.

In theory, the PPS is implemented on a Mica2 mote. However, as discussed in Section 3.1.3 individual requests to the PPS are impossible to measure in software with the system clock. TinyOS version 1.1 is used as it is the most strenuously tested. The code is compiled for the Mica2 and a static analysis of the disassembled code is performed to determine the number of instructions used and cycles required for execution. The disassembled code is produced with avr-objdump, a disassembler included with the distribution of TinyOS 1.1. Finally, a cycle count is determined from the ATMega128L datasheet [Atm04].

Another important metric of the PPS is the randomness and period of the proposed PRNGs. Many statistical tests exist to test randomness. These tests will be applied if the randomness of the algorithm is in question. Additionally, the period of most PRNGs can be verified empirically by generating the full sequence. Certain PRNGs, specifically CS-PRNGs, are specifically designed to make it computationally infeasible to generate the full sequence. These must be verified with a mathematical proof, but since it is not the aim of this thesis to create such an algorithm, this type of verification should not be needed.

3.9 Experimental Design

The EBS is designed to demonstrate the weakness of the RandomLFSR module, and there is no comparison being performed. Therefore the experiment is straightforward. Eight experiments are run to obtain a full factorial test of the time to break the encryption. As shown in Table 3, one factor of two levels and one factor of four levels are chosen for the EBS. Thus, two times four equals eight experiments. Each experiment for a given network size using a random number of rekeys is replicated 20 times totaling 80 experiments (1 * 4 * 20 = 80). When using zero re-keys each network size level is deterministic and thus each experiment only needs to be executed once. A total of 84 experiments are run to profile the performance of the EBS.

The PPS is designed to measure the performance of the enhanced PRNG compared to the LFSR-PRNG. The PPS is analyzed analytically requiring no experiments to be run.

3.10 Analysis and Interpretation of Results

The execution time for the EBS is averaged over the replications for each experiment. These are plotted on a graph and visually verified for linearity. The key space of each network size tested by the EBS does not proceed in a linear manner. That is to say, a network of size 6 does not contain twice as many keys as a network of size 3. This is due to the sub-maximal performance of RandomLFSR. However, the expected time for each network can be obtained by knowing *t*, the time to compute a private-public key pair. The value *t* is determined empirically via repeated measurements of a single private-public key computation on the target laptop. Each network size level forms its own category and the average of each category is calculated to verify that it is in the range of the expected value of $\frac{1}{2}tn$ with 90% confidence.

The PPS is an attempt to provide TinyOS with an alternative to the LFSR-PRNG. Metrics are obtained on its performance to give implementers an idea of the costs associated with the proposed PRNG. In addition, the randomness and period of the proposed PRNGs are presented and discussed.

3.11 Summary

Chapter III presented the experimental methodology for key generation in sensor networks. Goals and objectives are stated and assumptions and limitations are discussed. Section 3.1 states the objective of breaking EccM 2.0 with a brute-force attack. As a means to this end, this work must determine the average time to perform a private-public key operation on the experimental laptop. Section 3.1 also states the objective of

proposing an alternative to RandomLFSR at a similar cost in terms of execution time and memory requirements.

IV. Analysis and Results

4.1 Encryption Breaking System

The following subsections discuss the procedures used to discover the weaknesses in RandomLFSR and consequently EccM 2.0. They also discuss the EBS design, implementation, and performance. An analysis and interpretation of the results is presented alongside the discussion of the EBS performance.

4.1.1 RandomLFSR Analysis

The LFSR-PRNG of TinyOS, hereafter called RandomLFSR, was never designed to provide security primitives with random numbers. It is initialized with the mote's TOS_LOCAL_ADDRESS constant, which is assigned during configuration and program upload. This constant is the mote's network address and is used when routing traffic through the WSN. Thus it can be discovered simply by monitoring the packets generated by the mote.

TinyOS 1.1.0, and consequently RandomLFSR, are written in the nesC language [GLB03]. In order to implement the EBS, it was necessary to port the code to Java. The port also made it easier to quickly examine the sequence for any given seed and analyze the behavior of RandomLFSR. As mentioned, the mote ID also acts as the seed for RandomLFSR. The mote ID is a 16-bit unsigned integer and in theory can range from 0-2^{16}-1. However, in actual operation certain network addresses should not be used, such as 65535, or 0xFFFF in hexadecimal, and 126, or 0x007E in hexadecimal. These values represent the network broadcast address and Universal Asynchronous

Receiver/Transmitter (UART) address. This turns out to be irrelevant since there are no arbitrary restrictions on the seed to RandomLFSR, not even a seed of zero.

[Lev04] described the odd behavior of RandomLFSR, but did not go into details. To better understand RandomLFSR, the random number sequence was generated for the full range of possible seeds. Table 4 illustrates the various sequence lengths for different seeds. The first column lists the various sequence lengths produced by RandomLFSR, the second column lists the number of seeds that generate a sequence of the given length, and the last column lists the percentage of the total possible seed values (2^{16}). Most notable is that while none of the mote IDs produces a maximal sequence, some IDs produce sequences shorter than others.

Another interesting behavior is that for seeds in the range 0 thru 2^{16}, 31796 of them produce a sequence of 31796 before repeating, 21159 produce a sequence length of

Table 4. The Various Sequence Lengths Produced by TinyOS 1.1.0's RandomLFSR

Sequence Length	# of Seeds	Percentage
31796	31796	48.52%
21159	21159	32.29%
7471	7471	11.40%
2492	2492	3.80%
1110	1110	1.69%
345	345	0.53%
325	325	0.50%
301	301	0.46%
294	294	0.45%
195	195	0.30%
12	24	0.04%
8	8	0.01%
7	7	0.01%
6	6	0.01%
2	2	0.00%
1	1	0.00%

21159, 7471 produce a sequence of length 7471, and so on as is shown in Table 4. The reason for this behavior is easily understood once it is realized that RandomLFSR creates cyclical subgroups from set of all 16-bit integer seeds. The seeds generating a sequence length of 31,796 form a cyclic subgroup of $\{x \mid x \in \mathbb{Z}, 0 < x < 65536\}$. If one of the elements generating this sequence length is used as a seed, it creates the same random sequence as any other number in the group. This behavior of RandomLFSR implies that only 17 unique sequences are ever generated. This follows from the 16 rows in Table 4 plus one row (sequence length 12) has two subgroups generating the same sequence length, thus 16+1=17. One minor detail in the code of the LFSR prevents this fact from being exploited. A numerical mask that is different for each seed is used to alter the output of RandomLFSR. The mask is calculated from the initial seed by the formula $mask = 137 * 29 * (TOS\_LOCAL\_ADDRESS + 1)$. Documentation for why the mask is calculated in this manner could not be found, especially why the multipliers of 137 and 29 were chosen. This mask is applied by XORing the mask with the output of RandomLFSR before the output is returned to the caller. Therefore, while the underlying sequences generated by each group are the same, the sequence generated for each seed is not. This does not affect the distribution of RandomLFSR as shown in Figure 6, which shows the distribution of the sequence of numbers generated by a seed of zero.

4.1.2 EccM 2.0 Analysis

EccM 2.0 represents large integers using a byte array to hold the bits of the integer. This representation is commonly known as a "big integer" or *bigints*. For example, to represent 2^{163} using a byte array requires an array length of 21. Thus, the

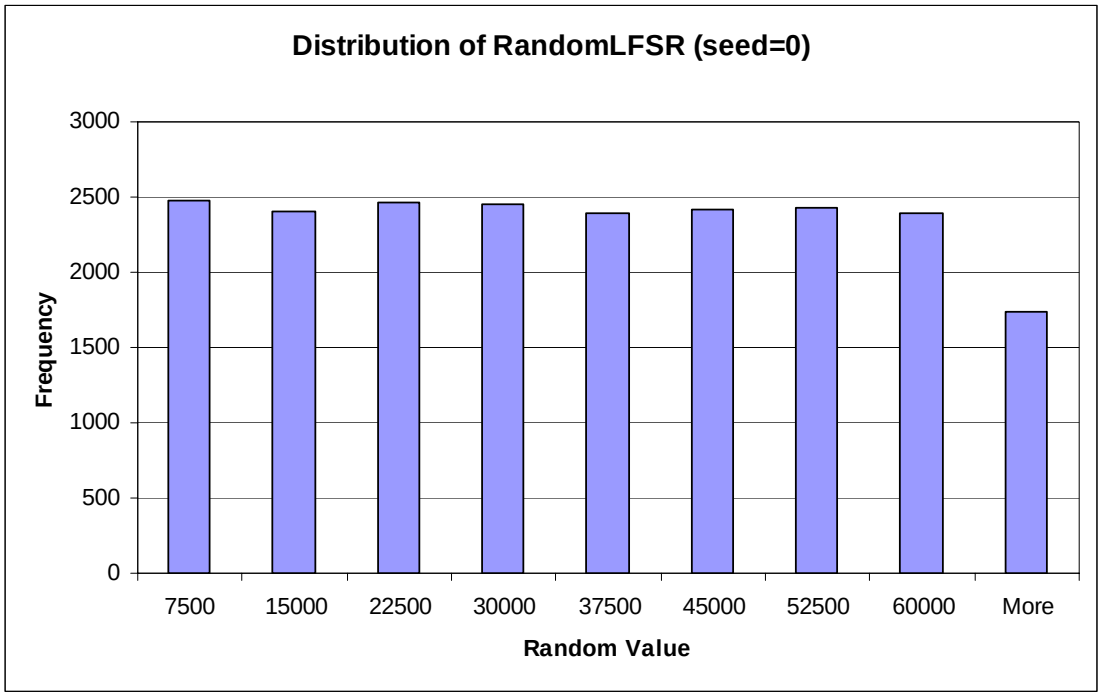

Figure 6. A Typical Distribution of the Numbers Produced by RandomLFSR

right-most element of the array represents the 8 least significant bits of the big integer. The left-most element (indexed at 0) represents the 8 most significant bits. EccM 2.0 actually uses an array of length 42 to handle overflow as the number is manipulated by the elliptic curve algorithms. Figure 7 shows the section of code used to generate the private key. The function *b_mod* is function used by the author of EccM 2.0 to perform modular arithmetic on *bigints*. It takes to two byte arrays representing *bigints* and the bit length of the *bigints*, uses the second *bigint* argument as the modulus, and places the result in the first *bigint* argument.

```
// privKeyA.s = random number in [0, 2^p);
for (i = NUMWORDS/2; i < NUMWORDS; i++)
  privKeyA.s[i] = (word_t) call Random.rand();

// privKeyA.s = privKeyA.s (mod params.r)
b_mod(privKeyA.s, params.r, NUMWORDS/2);
```

Figure 7. EccM 2.0 Algorithm for Generating the Private Key

It is the intention of this section of code to randomly select a number between 0 and *r*, the order of the elliptic curve, with equal probability. The order of the curve (*params.r* in the code) is a 49 digit number and is roughly given as 5.84e48, somewhere between 2^{163} and 2^{162} [Mal04]. However, because of the deficiencies of RandomLFSR it is known that at most 31,179 numbers can ever be generated in the best case. This is because the private key is created via repeated calls to the RandomLFSR module. If *n* is the length of the sequence and *S* is the sequence for a given mote, then the first call to RandomLFSR can be viewed as the starting point of some subsequence in *S*. There are only *n* such subsequences in *S* and therefore only *n* private keys will be generated from *S*.

The author of EccM 2.0 specifically chose a 163-bit key with the intention of providing 80 bits of security as per the recommendation of the NIST. However, in the best case, a little less than 15 bits of security ($2^{15} = 32768$) is provided by EccM 2.0. This fact leaves EccM 2.0 vulnerable to a brute-force attack. All that is needed to discover the private key for a given public key is to know the sequence used (i.e. know the seed).

4.1.3 Encryption Breaking System

With the RandomLFSR understood and EccM 2.0 exposed, it simply remains to design a system that faithfully duplicates the steps performed by EccM 2.0 to generate public keys. The Encryption Breaking System (EBS) is implemented in Java. The EBS is intended to run on a laptop with a mote connected via the serial port that is listening indiscriminately to packets transmitted over the air. Tools to forward packets from the mote to a connected PC are already written in Java and provided with the distribution of TinyOS 1.1.0, thus Java was a natural choice for the EBS. It is a simple matter to modify these tools to seamlessly pass packets to the EBS. In order to maintain a controlled experiment, an offline process generates all public keys used in the experiment and stores them in a database. When the simulation of EBS runs, a seed and public key are randomly selected from the database and artificially injected into the EBS. The encryption breaking program is triggered by the simulation in the same manner that an overheard packet is forwarded over the serial port. That is to say, they use the same method call.

The EBS now possesses the two things it needs to discover the private key via a brute-force attack, namely the mote's ID and the public key. It is simply a matter of time before the key is discovered. How much time is of concern because taking too much time renders the private key useless to the attacker since the mote may have rekeyed. For example, if it takes several months to discover the private key by trial and error, it is likely that the mote will have rekeyed itself by then. On the other hand, if the key can be

discovered in a trivial amount of time, then in order to remain secure, the mote must rekey itself at least as fast as the average case.

4.1.3.1 The Expected Performance of EBS

In order to verify the execution time of the EBS, the variables of equation 1, must be known. The key space, *n*, is easy enough to determine. The key space for each mote used in the experiment is known because it is simply the period of that motes' RandomLFSR sequence. However, the time to generate the private-public key pair, *t*, is not known. One possible way to determine this value is to measure the time it takes for the offline process to generate the public keys used within the experiment. The process stored these keys in text files that can be loaded by the simulation to randomly feed keys and IDs to the EBS. Table 5 shows that this requires 566,500 iterations. The sequence length of each of the mote IDs selected for experimentation was measured empirically by generating the entire sequence. As mentioned in Section 3.7, it is common practice to address motes in a network starting at 0 and counting up. Therefore, mote IDs 0-26 are used to obtain the numbers in Table 5. Consequently, 8 mote addresses in the range 0-26 have a sequence length of 31,796, 14 have a length of 21,159, etc. The key space for all

Table 5. Sequence Length of First 27 Sequential Mote IDs

Sequence Length	# of Motes	Total Keys
31796	8	254,368
21159	14	296,226
7471	2	14,942
345	1	345
325	1	325
294	1	294
Total	**27**	**566,500**

27 motes is simply the summation of the right column in Table 5, which is the sequence length multiplied times the number of motes.

This presented the opportunity to measure the execution time of the laptop when generating the keys required. On average, the laptop took approximately 97.14 msecs to compute a public key with a standard deviation of 6.87 and 90% confidence interval of ±0.015 msecs. Appendix B discusses how the average is derived.

The sheer size of the task created additional difficulties. The public keys are selected "on the fly" while the simulation is running. However, attempting to load the 27 text files containing the public keys causes the JVM to throw an OutOfMemoryError error due to the average file size of 5.5 MB. Therefore, as the simulation runs, the file for the target mote is loaded and a public key is randomly selected. This implies that a small amount of overhead in terms of file loading time is introduced for each mote that would not be present in the actual system. The average load time for each file is shown in Figure 8. To put things in perspective, Table 6 shows the expected performance for each network size and the overhead introduced. Appendix C provides the data for the numbers used to derive the average load time.

4.1.3.2 Results and Analysis of EBS Performance

To simulate attackers present during a DSN deployment, four experiments on different network sizes used zero rekeys. If an attacker is present at deployment he will overhear the first broadcast of any mote in the network. This leaves EccM 2.0 extremely vulnerable to attack due to its straightforward startup. An attacker simply needs to generate the first pair of keys in the sequence to find the private key. Table 7 illustrates

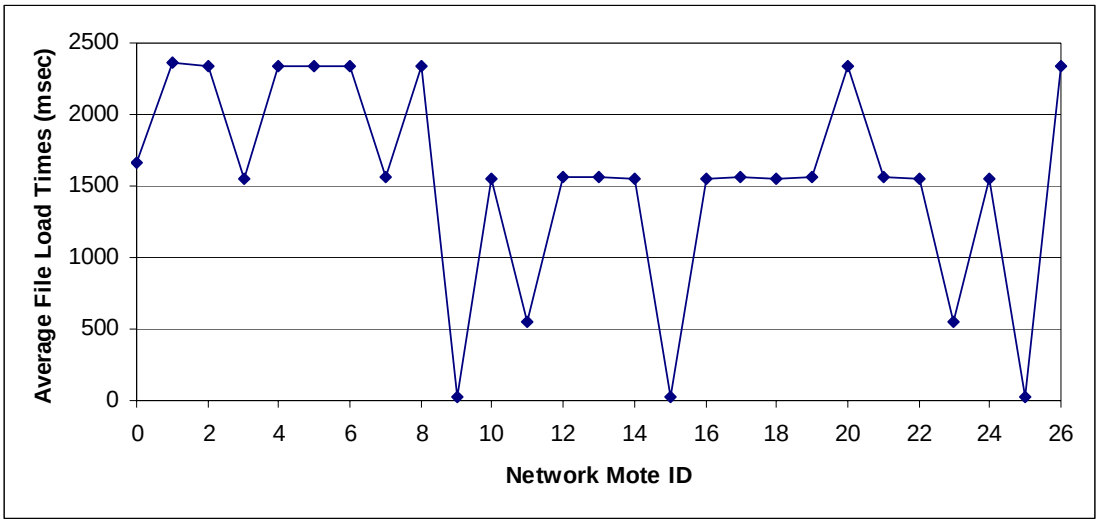

Figure 8. Average Times for Loading Key File for Each Mote

Table 6. Expected Time to Identify a Private Key Versus Average File Load Time

Network Size	Expected Time	Average File Load Time	Ratio of Load/Expected
3	68 min, 36 sec	6 sec	0.0015
6	137 min, 12 sec	12 sec	0.0015
9	205 min, 48 sec	18 sec	0.0015
27	458 min, 34 sec	41 sec	0.0015

the performance of the four experiments. The first column lists the network sizes used in the experiments and the second column lists the running time to find the private keys of all the motes in the network. The measured times are higher than the expected time of about 97 msecs per mote. This is partly explained by the file loading times listed in the third column of Table 6. However, there is still a significant gap, which is assumed to be due to program overhead at startup.

Table 7. Measured Time of the EBS to Identify Private Keys with Zero Rekeys.

Network Size	***Measurement (msecs)***
3	7016
6	13109
9	19656
27	44235

In contrast to an attacker being present at DSN deployment, it is possible for EccM 2.0 to rekey an arbitrary number of times. This means that any key in the key space could be in use at any given time. To simulate this, a random key is selected from the key space for each mote and injected into the EBS. The size of the key space for each network size is dependent upon the mote IDs used for each network. As discussed in Chapter 3, it is common practice to sequentially number the mote IDs starting at 0. This results in a key space of 84,751 for 3 motes, 169,502 for 6 motes, 254,253 for 9 motes and 566,500 for 27 motes. Figure 9 plots the sequence length of each mote ID over the cumulative key space for a network consisting of the greatest assigned network address. The x-axis represents the mote ID, the left y-axis is the sequence length of the particular mote ID, and the right y-axis is the size of the network assuming motes are addressed sequentially. For example, a network of 3 motes consists of motes addressed 0, 1, and 2. The key space for this network is the point labeled 2 on the x-axis and is the summation of the individual mote's key spaces. As discussed in Section 4.1.2, the key space of a network is essentially equal to the sequence lengths generated by RandomLFSR, which is based on the mote's network address.

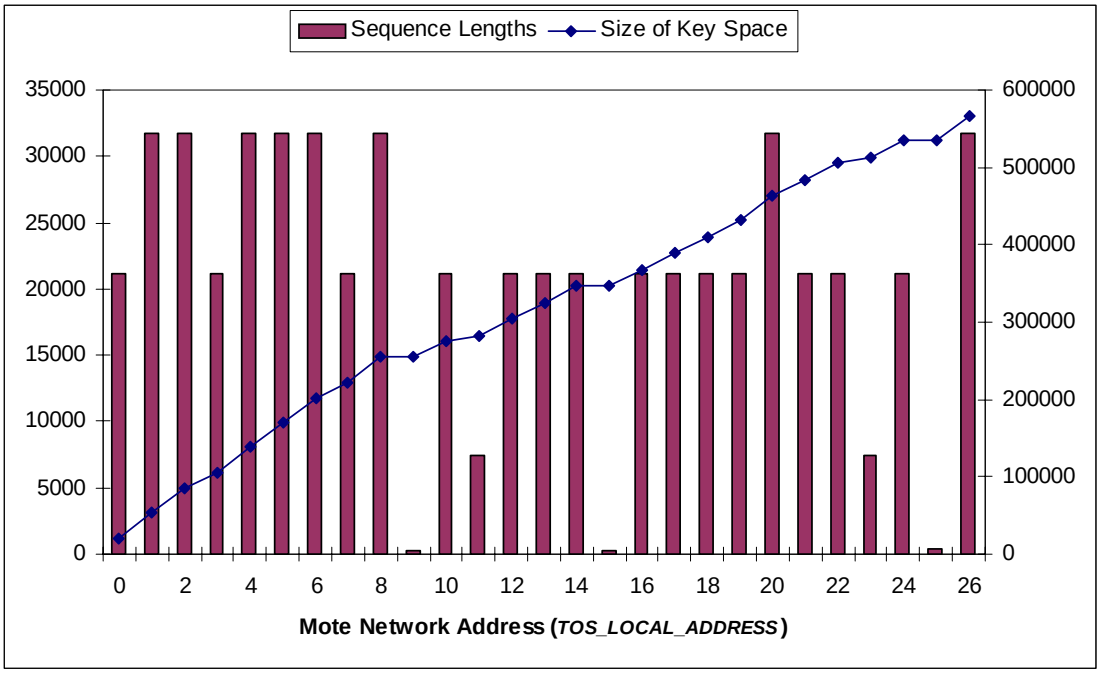

Figure 9. The Correlation of RandomLFSR Sequence Lengths and Size of the Key Space Assuming Motes are Addressed Sequentially

Table 8 summarizes the results from the experiments for the different sized networks. The average time in the second column is the measured time from each experiment averaged over 20 measurements. The expected time in column three is

Table 8. The Average Time for the EBS to Identify All Keys in a Network of a Given Size Assuming an Arbitrary Number of Rekeys

Network Size	**Average Time**	**Expected**	**% Error**	**Standard Deviation**
3	74 mins, 36 secs	68 mins, 36 secs	8.82%	24 mins, 42 secs
6	138 mins, 11 secs	136 mins, 12 secs	0.72%	31 mins, 42 secs
9	222 mins, 16 secs	205 mins, 49 secs	8.00%	49 mins, 29 secs
27	455 mins, 3 secs	458 mins, 34 secs	0.77%	138 mins, 34 secs

computed using equation 1, with 97.14 milliseconds as *t*. The key space, *n*, is determined empirically, since it is a simple matter to generate the random sequence for each mote address. The percent error is the difference of the average measured time and the expected time divided by the expected time. Figure 10 shows these results graphically with the error bars giving the 90% confidence interval for each point. The expected time is plotted for each network size. Obviously it closely resembles the line in Figure 7 since it is simply the key space multiplied times the constant ½*t*. Thus, it appears that the

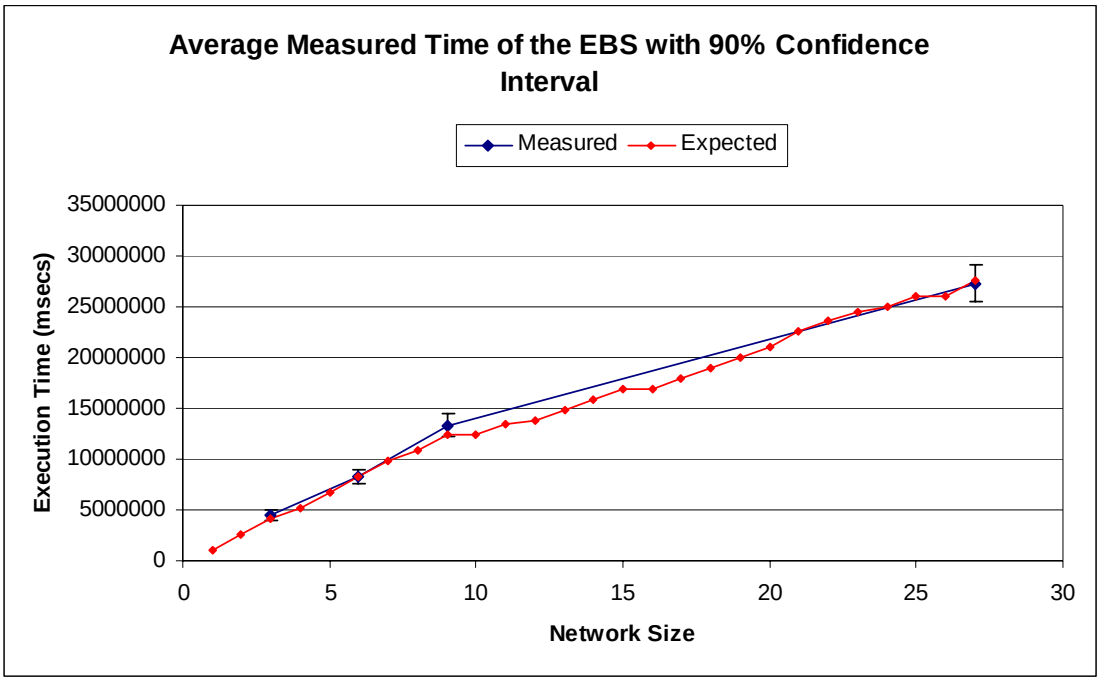

Figure 10. The Measured Time to Find Private Keys Versus Expected Time

performance of the EBS supports the expected time of 97.14 milliseconds per private-public key operation.

The result of all this is that for 31,796 potential mote IDs, the time that EBS can discover the private key on average is:

$$\frac{1}{2}tn = \frac{1}{2} \times 97.14 \text{ milliseconds } \times 31796$$
$$= 1544331.72 \text{ milliseconds}$$

or in minutes:

$$= \frac{1544331.72 \text{ milliseconds}}{60000 \frac{\text{milliseconds}}{\text{minute}}}$$
$$= \boxed{25.738862 \text{ minutes}}$$

This means that in order for EccM 2.0 to remain secure, the mote must transition a private-public key pair to a deactivated state 25 minutes after its initial use. This is at a minimum; a prudent design would allow the key to remain in an active state at most 15 minutes. Of course, this assumes that only motes with the maximum period are used. Table 9 shows the expected time to identify a private key for all sequence lengths generated by RandomLFSR. The times in the second column are computed using equation 1. The last row groups sequence lengths of 12 or less together and specifies that it takes 600 milliseconds or less to break sequence lengths of 12 or less. How this affects the longevity of the WSN is application dependent. For example, assuming the batteries provide 2200 mAh at 3V and considering that the Mica2 processor requires 8 mA to operate and that a rekey operation on the Mica2 takes 68.53 seconds, then the Mica2 can calculate approximately:

Table 9. The Expected Time of the EBS to Find a Private Key Given a Mote Producing the Listed Sequence Length

Sequence Length	Expected Time to Identify Private Key
31796	25.74 minutes
21159	17.13 minutes
7471	6.05 minutes
2492	2.02 minutes
1110	53.91 seconds
345	16.76 seconds
325	15.79 seconds
301	14.62 seconds
294	14.28 seconds
195	9.47 seconds
12 *or less*	< 600 milliseconds

$$\frac{2200 \text{ milliamp-hours} \times 3600 \dfrac{\text{seconds}}{\text{hour}}}{8 \text{ milliamperes} \times 68.53 \dfrac{\text{seconds}}{\text{rekey}}} = \frac{7920000 \text{ milliamp-seconds}}{548.24 \dfrac{\text{milliamp-seconds}}{\text{rekey}}}$$

$$= \boxed{14446 \text{ rekeys}}$$

before exhausting its batteries. Assuming four rekeys per hour allows the Mica2 to last for about 151 days. Of course, this is the best case scenario and concedes that the Mica2 does nothing else in those 151 days except rekey itself.

In general, it is difficult to estimate the longevity without an application. If the key is expired every 15 minutes, or four rekeys per hour, the processor would require a duty cycle of:

$$4 \times \frac{68.53 \dfrac{\text{seconds}}{\text{hour}}}{3600 \dfrac{\text{seconds}}{\text{hour}}} \times 100 = 7.614\%$$

Of the few applications actually published, most constrain the processor's duty cycle to no more than 5.8% [MPS02]. Since the duty cycle required for secure operation is greater than what is allowed by the application, a compromise must be made somewhere. For example, it is possible to design a DSN that operates only 15 minutes every hour. In fact, many designs may already do this in the interest of conserving power. However, this constraint is an unnecessary burden on a DSN engineer since better alternatives to RandomLFSR exist in the literature.

4.2 PRNG Performance System

The following subsections discuss alternatives to the RandomLFSR PRNG of TinyOS. Two alternatives are examined. The first is a maximal LFSR-PRNG adapted from [Sch96]. The second is a MLCG-PRNG adapted from [PaM88] and distributed with the beta of TinyOS 2.0. Their performance is analyzed in terms of computation cycles, memory requirements and sequence lengths. Due to their fundamentally different design, a direct comparison is inappropriate. However, a system's view is taken when analyzing the impact of each PRNG on the performance of EccM 2.0 that clearly shows the best alternative.

4.2.1 A Maximal Linear Feedback Shift Register

One possible way to overcome the deficiencies of RandomLFSR is to create a maximal LFSR and seed it with a secret number. The literature is replete with examples of how to create a maximal LFSR. The linear nature of LFSR-PRNGs however make this an even weaker PRNG then RandomLFSR. Since the next number of an LFSR, x_{i+1}, is dependent on the previous number, x_i, a random sequence produced for a given

maximal tap sequence forms a linear chain, x_0, x_1, x_2, …, x_{n-1}, x_n, where n is the period of the LFSR. The result of this behavior is that no matter which value is chosen as the seed it will always produce the same sequence. All that is needed to break EccM 2.0 in this case is produce 2^{16}-1 public keys. Each mote produces the same set of keys. This set of keys could be generated off-line and reduce breaking EccM 2.0 to a reverse lookup table.

RandomLFSR avoids this pitfall by creating a mask based on the seed value as discussed in Section 4.1.1. The same approach can be used for a maximal LFSR and thus produce 2^{16} unique sequences, where a seed of zero produces the base sequence. The maximal property of the LFSR is not affected by this modification, which was verified empirically by generating the sequences for all possible seed values. The code for this maximal LFSR, hereafter referred to as MaximalLFSR, is shown in Figure 11. The main body of the function is taken directly from [Sch96] and implemented in the nesC language. The full code can be viewed in Appendix F. *shiftReg* is the 16-bit shift register of MaximalLFSR and is essentially the state of the LFSR. The atomic statement prevents system interrupts and ensures other tasks cannot alter the state of the MaximalLFSR. *endbit* is a Boolean value representing the exclusive-or of the 16^{th}, 5^{th}, 3^{rd}, 2^{nd}, and 1^{st} bits and determines the "feedback" of the LFSR. If *endbit* evaluates to true, *shiftReg* is shifted right by one and a 1 is placed in the left-most bit. If it evaluates to false, *shiftReg* is shifted right by one and a 0 is placed in the left-most bit by definition of the right-shift operation in nesC.

The design is slightly modified from [Sch96] in that instead of outputting a single bit, the entire state of MaximalLFSR is XORed with *initseed*, which is simply the initial

```
async command uint16_t Random.rand() {
  bool endbit;
  uint16_t tmpShiftReg;

  atomic {
    tmpShiftReg = shiftReg;
    endbit = ((tmpShiftReg >> 16) ^ (tmpShiftReg >> 5) ^
              (tmpShiftReg >> 3)  ^ (tmpShiftReg >> 2) ^
              (tmpShiftReg)) & 0x0001;
    if (endbit)
      tmpShiftReg = (tmpShiftReg >> 1) | 0x8000;
    else
      tmpShiftReg = tmpShiftReg >> 1;
    shiftReg = tmpShiftReg;
  }
  return tmpShiftReg ^ initseed;
}
```

Figure 11. The MaximalLFSR Function for Generating the Next Random Number

seed of MaximalLFSR, and returned. As discussed in [Sch96], LFSR-PRNGs are designed to produce random bits. To produce random numbers from LFSR-PRNGs, [Sch96] suggests repeatedly calling the random function and ORing the bits together to form the desired length integer. Thus, given that an *n*-bit number is desired and a *n*-bit LFSR is used, the resulting value after *n* calls is simply the state of the LFSR *n* calls prior. Therefore, for a 16-bit PRNG such as MaximalLFSR it seemed logical to simply return the current state of the LFSR. This approach succeeds at thwarting a brute force attack because knowledge of the private key cannot be gained from the public key. In other words, while knowledge of the private key reveals information about the underlying random sequence, all the cryptanalyst has access to is the public key. Thus, when using

MaximalLFSR, the EBS must generate $2^{16} * (2^{16}-1) * ½$ public keys on average to break the encryption. A successful attack by the EBS would take on average:

$$\frac{1}{2} \times 97.14 \ \frac{\text{milliseconds}}{\text{key}} \times \left(2^{16}\left(2^{16}-1\right)\right) \text{ keys} = 208{,}603{,}378{,}483.2 \text{ milliseconds}$$

or in years:

$$\frac{208{,}603{,}378{,}483.2 \text{ milliseconds}}{1000 \times \frac{3600 \text{ second}}{1 \text{ hour}} \times \frac{24 \text{ hours}}{1 \text{ day}} \times \frac{365.25 \text{ days}}{1 \text{ year}}} = \boxed{6.61 \text{ years}}$$

However, EBS fails in this case by the definition of failure. Using equation 2 results in a duty cycle for the Mica2 of:

$$\frac{2 \times \left(\frac{3{,}600{,}000 \frac{\text{milliseconds}}{\text{hour}}}{97.14 \ \frac{\text{milliseconds}}{\text{key}}} \right)}{2^{16}\left(2^{16}-1\right) \text{ keys}} \times 1.9\% = \boxed{3.27 \times 10^{-5}\% < 0.1\%}$$

Unfortunately, MaximalLFSR does not produce random sequences. While no statistical tests were run on the output of MaximalLFSR, a cursory analysis shows that the output is predictable for internal states that are powers of two. For example, suppose *shiftReg* equals 2^{14}, or 0x4000 in hexadecimal. In this case, *endbit* evaluates to 0 and the else branch is executed, which simply left-shifts the value of *shiftReg* by 1. In other words, it divides 2^{14} by 2 giving 2^{13}, or 0x2000. This pattern continues for 0x1000, 0x0800, 0x0400, 0x0200, and so on, until the internal state becomes 2^5 and *endbit* evaluates to 1. While the exclusive-or operation on the output transforms this sequence, this pattern is still easily identified. For example, suppose *initSeed* equals 1, then the pattern is the same, except that one is added to each element of the sequence. The

exclusive-or changes the underlying pattern in a consistent, predictable way, which is readily detected under close inspection. This in turn, reveals the initial seed and allows the output to be predicted.

4.2.2 TinyOS 2.0's Multiplicative Linear Congruential Generator

The beta version of TinyOS 2.0 was released in November 2005 [Tin05]. This release came packaged with two random number generators, the original RandomLFSR and a new PRNG proposed in [Lev04]. The new PRNG is based upon the Park-Miller minimum standard proposed in [PaM88] and is a multiplicative linear congruential generator (MLCG). Researchers extensively tested the output of this MLCG for randomness [CoM67, FiM86, Hoa76, Knu81] and it is generally accepted to be a good PRNG [PaM88]. While MLCGs are excellent choices for simulations, they prove to be a poor choice for cryptographic systems as they are predictable [Sch96]. With that said, seeding TinyOS 2.0's MLCG with a secret seed (presumably loaded during configuration) succeeds at thwarting the attack defined in this thesis because the sequence generated by the MLCG is maximal with a period of 2^{31}-1. For the EBS to successfully identify the private key would require on average:

$$\frac{\frac{1}{2}\times 97.14\ \frac{\text{milliseconds}}{\text{key}}\times\left(2^{31}-1\right)\ \text{keys}}{1000\frac{\text{milliseconds}}{1\ \text{second}}\times\frac{3600\ \text{second}}{1\ \text{hour}}\times\frac{24\ \text{hours}}{1\ \text{day}}\times\frac{365.25\ \text{days}}{1\ \text{year}}}=\boxed{3.305\ \text{years}}$$

RandomMLCG also manages to causes the EBS to fail by only requiring a duty cycle of:

$$\frac{2\times\left(\frac{3{,}600{,}000\ \frac{\text{milliseconds}}{\text{hour}}}{97.14\ \frac{\text{milliseconds}}{\text{key}}}\right)}{\left(2^{31}-1\right)\ \text{keys}}\times 1.9\% = \boxed{6.56\times10^{-5}\% < 0.1\%}$$

4.2.3 Performance of Alternatives

Sections 4.2.1 and 4.2.2 examine the behavior of each of the proposed alternatives for uniformity and randomness. Also of interest is the cost of implementation on the Mica2, particularly as it relates to computation time. Analysis of the compiled binaries reveals the number of instructions and cycles required to implement both RandomLFSR and MaximalLFSR. It is shown that the number of branch instructions of RandomMLCG increases significantly compared to both RandomLFSR and MaximalLFSR, which hinders an exact static analysis of the cycle requirements. Simulation software could have been used to obtain a more precise profile of RandomMLCG than is presented in this section. However, the main objective of this analysis is to show that one alternative is superior to the other, which is clearly evident with the analysis presented. Furthermore, the analysis of RandomLFSR and MaximalLFSR is as precise as possible and simulation of these modules would not have added any new information.

Table 10 shows the results of the analysis. The instruction count, cycles required for execution, total number of branch instructions in the assembly, and memory requirements in bytes are listed for each PRNG. The third column lists the number of cycles required for execution of a single function call. Branch statements result in different cycle requirements depending on the internal state of the PRNG and are represented by multiple values. RandomMLCG has branch statements within branch

statements. There are two main branches of execution, which are presented with approximate values since nested branch statements result in different cycle counts.

Table 10. A Static Analysis of Alternative PRNGs Versus RandomLFSR

PRNG	Instruction Count	Cycles	Branch Instructions	Size (bytes)
RandomLFSR	21	20, 23	1	46
MaximalLFSR	41	146, 148	5	86
RandomMLCG	540	~916, ~1024	34	1108
Optimized MaximalLFSR	74	71, 73	1	152

Four of the five branch statements in MaximalLFSR are used in an iterative loop that evaluates a register that is decremented each pass through the loop. Optimization of the assembly code could eliminate the execution of 75 cycles resulting in a requirement of 71 or 73 cycles. Of course, this is at a cost of 66 bytes of program memory. See Appendix E for a discussion of the optimization used to determine these figures.

4.3 Summary

This chapter consists of two main sections. Section 4.1 discusses the EBS and the work carried out to answer the questions put forth in Section 3.1.1. Section 4.1.1 presents the analysis performed on RandomLFSR to determine its behavior. Section 4.1.2 shows how the short period of RandomLFSR leaves EccM 2.0's key generation algorithm vulnerable to a brute-force attack. Finally, Section 4.1.3 presents the steps taken to exploit this vulnerability and determines the expected time to identify a private key given a mote's address and public key.

Section 4.2 discusses the PPS and the analysis necessary to meet the goals laid out in Section 3.1.1, namely the goal of finding an alternative capable of thwarting a brute-force attack on EccM 2.0. Sections 4.2.1 and 4.2.2 presents PRNGs based on LFSR and LCG designs, respectively. Both alternatives succeed at thwarting the brute-force attack defined in this work. Section 4.2.3 presents the cost of implementation on the Mica2 for each alternative as well as the costs associated with RandomLFSR.

V. Conclusions and Recommendations

5.1 Restatement of the Problem and Conclusions

Chapter 3 asks the following questions:

1. Can the weaknesses of the LFSR-PRNG in TinyOS be exploited to break the public key cryptography and, if so, how fast can it be done?
2. Will the rate of key compromise exceed the mote's capacity to rekey?

Chapter 4 shows that the LFSR-PRNG used in TinyOS exposes the elliptic curve cryptosystem proposed in [Mal04,MWS04] to a brute-force attack. The average time to discover the private key for a given mote is 97.14 milliseconds. Practically speaking, this requires the mote to rekey once every 15 minutes. This equates to a 7.6% duty cycle. While the needs of DSN applications vary widely, this is an additional constraint on an already severely constrained environment. There is a better solution.

Another stated goal of this work is to produce an alternative to the RandomLFSR module in TinyOS that thwarts the brute-force attack defined in this thesis. One alternative, MaximalLFSR, was proposed by the author. Another alternative, RandomMLCG, comes packaged with the beta release of TinyOS 2.0. Although both fail requirements of a cryptographically secure PRNG due to their linear nature, they succeed at defeating the brute-force attack when the seed is kept secret. MaximalLFSR does this at a much lower cost than RandomMLCG while requiring significantly more computation time on the part of the attacker (6.6 years versus 3.3 years).

5.2 Contributions and Significance

The main contribution of this thesis is the analysis of the current RandomLFSR module distributed with the current version of TinyOS and in the beta release of TinyOS 2.0. This module is used throughout the operating system and in a wide array of applications. While it is likely that many applications using RandomLFSR are tuned to operate with its quirkiness, certain seeds are shown to produce sequences of only a few numbers. At the very least, users of TinyOS should be made aware of these limited seeds and avoid them. Even better, RandomLFSR should be replaced with MaximalLFSR, as it performs comparably in terms of execution time and memory requirements.

A secondary contribution includes the demonstration of a brute-force attack on EccM 2.0, an implementation of ECDLP on the Mica2. Although it is a simplistic attack that is easily thwarted, it examines the threat of such an attack given a computationally superior attacker that future DSNs could face in the field. A determined attacker could transmit public keys overheard via a laptop connected to a mote to a more powerful computing system. The attack described in this thesis can be easily modified to work in parallel and use the power of distributed computing to break the cryptosystem more quickly.

5.3 Recommendations for Future Research

In the process of answering the questions put forth in this thesis, many more questions surfaced. Perhaps most tantalizing is the possibility that the public keys could somehow be correlated to the underlying sequence used to generate the private keys. The structures of RandomLFSR and MaximalLFSR are very simplistic and quite a bit of

mathematical theory could be applied to discover nonrandom properties. If these properties reveal themselves in the key space of the mote, perhaps they could be generalized to apply to all motes. Thus, the public key could be used to determine which sequence is used and where in the sequence the key is generated from. This information could then be used to generate the private key.

EccM 2.0 itself still requires much work before it is complete. A generalized protocol for exchanging keys in an arbitrarily large network must be devised. In addition, the shared key is actually a point on the elliptic curve used in EccM 2.0. To use this shared secret in a symmetric key cipher, a secure mechanism is needed for transforming this shared secret into a secret key suitable for use in TinySec or other symmetric cryptosystem. One such mechanism is a cryptographic hash function. These can also be used in combination with a counter to provide secure key generation. An efficient implementation of a cryptographic hash function in nesC can serve dually as a source of random bits for cryptographic keys and a secure hash of the shared key produced by EccM 2.0.

The brute-force attack described in this thesis works well on relatively small key spaces. However, a key space of only 2^{25} requires roughly 18 days of computation, assuming 97.14 milliseconds per private-public key operation, to discover one key. Obviously, it does not scale well because it is limited to one laptop with a single processor. A determined attacker with greater computational power can easily overcome the key space provided by MaximalLFSR. Indeed, using a system similar to the one used to crack DES, the entire key space of a WSN using MaximalLFSR could be computed in

less than a second. This information could then be used to simply lookup the private key of the public key in use. This begs the question of how large the key space must be to thwart such an attacker. WSNs are extremely susceptible to brute-force attacks that look to overpower the resources of the individual mote.

5.4 Summary

This work presents a brute-force attack on an implementation of ECDLP on TinyOS. The attack exploits the short period of the RandomLFSR module of TinyOS. It demonstrates an average compromise time of 25 minutes for the longest sequence produced by RandomLFSR. Over 50% of possible mote addresses lead to significantly shorter compromise times. The possibility of distributing the attack over multiple machines is an area for future research that could lead to even shorter compromise times.

Two alternatives to RandomLFSR are examined that can thwart the brute-force attack presented in this work. RandomMLCG, the multiple linear congruential generator distributed with TinyOS 2.0 beta, offers a much longer period of 2^{31}-1, but at a significantly greater cost in terms of computation time and program memory. MaximalLFSR, an alternative based on the well-known mathematics of LFSRs, performs more closely to RandomLFSR and roughly doubles the length of the period provided by RandomMLCG.

Appendix A: Derivation of Equation Two

Equation 2 in Section 3.3, page 36, is given as:

$$\frac{2r}{n} \times 1.9 = D$$

In part, this is derived from the fact that one rekey per hour on the Mica2 results in a duty cycle of 1.9%. Thus, the constant 1.9 in equation 2. It remains to determine a formula for calculating the expected number of keys compromised per hour. Equation 1 gives the expected time for one key compromise as:

$$\frac{1}{2}tn$$

where *t*, is the time for one private-public key operation and *n* is the size of the key space. Thus, if *t* is in hours, the expected number of key compromises per hour is simply the multiplicative inverse of equation 1:

$$\frac{2}{tn} \qquad (3)$$

Equation 3 must be in hours. To avoid confusion a new variable, *r*, is defined as the number of key trials per hour and is given as:

$$r = \frac{1}{t}$$

where *t* is in hours. Substituting *r* in for *t* in equation 3 gives:

$$\frac{2}{\frac{1}{r}n} = \frac{2r}{n}$$

which is the first term in equation 2.

Appendix B: Computing the Average Private-Public Key Operation

The computation of the key spaces for the first 27 sequential mote IDs provided ample opportunity to measure the time required to compute a single private-public key pair. In fact, exactly 566,500 private-public key pairs are computed. Computing the key pair involves two function calls, *generatePrivateKey(int index, byte[] arrSequence)* and *generatePublicKey(byte[] privKey)*. See Appendix X, pages XX and XX, respectively. The system clock is polled before the first call and after the second call and the difference is taken as the time to compute the key pair. This result is stored along with the public key in a text file for retrieval at a later time. It proved easier to write a program to compute the average then to load 566,500 measurements in to Excel. The program produced the output displayed in Figure 12.

Interestingly, if the formula for computing the variance is done by the program, it throws an exception because a non-terminating decimal expansion occurs when the JVM

Average	97.14295675198588
Sum	5.5031485E7
Sum of Squares	5.372645861E9
Count	566500
Measurement	Frequency
78	5591
79	715
93	107196
94	323739
109	81218
110	47989
125	42
140	4
141	6

Figure 12. Output of Program for Calculating Statistics of Time to Compute Private-Public Key Operations

tries to represent one of the terms of the equation as a double. Consequently, this data was then used to perform the following computations by calculator:

$$
\begin{aligned}
s^2 &= \frac{n\sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2}{n(n-1)} \\
&= \frac{566500 * 5372645861 - 55031485}{566500 * 566499} \\
&= 47.152 \\
s &= \sqrt{47.152} \\
&= 6.86842 \\
CI &= \pm z_{1-\alpha/2} \frac{s}{\sqrt{n}} \\
&= \pm 1.645 \frac{6.87}{\sqrt{566500}} \\
&= \pm 0.015
\end{aligned}
$$

Appendix C: Data Tables

Table 11. Statistics for Average File Load Times

Mote ID	Average	Sum	Variance	Standard Dev	90% CI
0	1665.6	16656	2260.71	47.547	20.7943
1	2362.5	23625	3398.5	58.297	25.4956
2	2343.7	23437	430.68	20.753	9.0761
3	1554.7	15547	600.01	24.495	10.7128
4	2339.1	23391	709.88	26.644	11.6524
5	2339	23390	555.78	23.575	10.3103
6	2339.2	23392	547.29	23.394	10.2313
7	1559.3	15593	693.34	26.331	11.5159
8	2340.7	23407	696.01	26.382	11.5380
9	25	250	60	7.746	3.3876
10	1554.7	15547	600.01	24.495	10.7128
11	546.8	5468	160.4	12.665	5.5389
12	1556.3	15563	725.57	26.936	11.7804
13	1560.9	15609	824.99	28.723	12.5616
14	1553.1	15531	646.32	25.423	11.1185
15	21.9	219	68.54	8.279	3.6208
16	1553.1	15531	489.21	22.118	9.6732
17	1559.4	15594	757.6	27.525	12.0377
18	1554.7	15547	784.23	28.004	12.2474
19	1557.8	15578	596.4	24.421	10.6805
20	2340.6	23406	759.16	27.553	12.0500
21	1559.4	15594	590.49	24.300	10.6274
22	1554.7	15547	767.12	27.697	12.1131
23	548.4	5484	133.16	11.539	5.0466
24	1553.1	15531	339.66	18.430	8.0601
25	26.6	266	57.16	7.560	3.3064
26	2339.1	23391	386.1	19.649	8.5935

Table 12. Statistics for Network Size = 3

Mean	4479343.05
Standard Error	331548.9481
Median	4516843.5
Standard Deviation	1482731.972
Sample Variance	2.19849E+12
Minimum	2098188
Maximum	7655797
Sum	89586861
Count	20
Confidence Level (90%)	573292.1584

Table 13. Statistics for Network Size = 6

Mean	8291583.7
Standard Error	425433.7989
Median	7802117
Standard Deviation	1902597.789
Sample Variance	3.61988E+12
Minimum	5978250
Maximum	13374094
Sum	165831674
Count	20
Confidence Level (90%)	735631.5327

Table 14. Statistics for Network Size = 9

Mean	13336702.35
Standard Error	664054.1218
Median	12922953
Standard Deviation	2969740.314
Sample Variance	8.81936E+12
Minimum	8710234
Maximum	19740094
Sum	266734047
Count	20
Confidence Level (90%)	1148237.758

Table 15. Statistics for Network Size = 27

Mean	27303081.15
Standard Error	1054178.669
Median	27383156.5
Standard Deviation	4714430.33
Sample Variance	2.22259E+13
Minimum	19027094
Maximum	35832671
Sum	546061623
Count	20
Confidence Level (90%)	1822814.906

Table 16. Expected Performance for Sequential Network Sizes

Mote ID	Cumulative Key Space	½ *tn, t=97.14* (msecs)
0	21159	1027692.63
1	52955	2572024.35
2	84751	4116356.07
3	105910	5144048.7
4	137706	6688380.42
5	169502	8232712.14
6	201298	9777043.86
7	222457	10804736.49
8	254253	12349068.21
9	254578	12364853.46
10	275737	13392546.09
11	283208	13755412.56
12	304367	14783105.19
13	325526	15810797.82
14	346685	16838490.45
15	346979	16852770.03
16	368138	17880462.66
17	389297	18908155.29
18	410456	19935847.92
19	431615	20963540.55
20	463411	22507872.27
21	484570	23535564.9
22	505729	24563257.53
23	513200	24926124
24	534359	25953816.63
25	534704	25970573.28
26	566500	27514905

Appendix E: Optimization of Assembly

Figure 12 presents the assembly code of MaximalLFSR. As Table 9 states, it consists of 5 branch instructions, located at addresses 0x1A4, 0x1AE, 0x1BC, 0x1DE, and 0x1DC. Addresses 0x1DE and 0x1DC are relative jumps and are required for proper execution. The *brne* instructions on the other hand are part of a loop that operates similarly to a do-while. For instance, the instructions in the address space 0x19C-0x1A4 constitute the first loop. The *ldi* instruction represents the initial condition and loads the constant 16 into the register *r26*. The next two instructions constitute the loop's work. The *dec* instruction decrements *r26* by one. The *brne* instruction evaluates *r26* to see if it equals zero. If not, the program jumps to the start of the loop at 0x19E at a cost of 2 CPU cycles. This occurs 15 more times until the final *dec* instruction sets *r26* to zero and the *brne* instruction evaluates to false at a cost of one cycle. This loop ultimately requires 80 cycles, 1 for the initial condition, 5 for each execution of the loop when *brne* evaluates to true (5*15=75), and 4 for the final loop iteration.

Alternatively, the loops could be manually unrolled and the work portion of the loop could be repeated 16 times. This increases the number of instructions the program contains resulting in a larger memory requirement for the program. However, the unrolling of the loop decreases the number of instructions that it executes, particularly the costly branch instruction. This essentially halves the execution time of MaximalLFSR.

```
194:   2f b7        in      r18, 0x3f       ; 63
196:   f8 94        cli
198:   62 2f        mov     r22, r18
19a:   ac 01        movw    r20, r24
19c:   a0 e1        ldi     r26, 0x10       ; 16
19e:   36 95        lsr     r19
1a0:   27 95        ror     r18
1a2:   aa 95        dec     r26
1a4:   e1 f7        brne    .-8             ; 0x19e
1a6:   f5 e0        ldi     r31, 0x05       ; 5
1a8:   96 95        lsr     r25
1aa:   87 95        ror     r24
1ac:   fa 95        dec     r31
1ae:   e1 f7        brne    .-8             ; 0x1a8
1b0:   28 27        eor     r18, r24
1b2:   ca 01        movw    r24, r20
1b4:   e3 e0        ldi     r30, 0x03       ; 3
1b6:   96 95        lsr     r25
1b8:   87 95        ror     r24
1ba:   ea 95        dec     r30
1bc:   e1 f7        brne    .-8             ; 0x1b6
1be:   28 27        eor     r18, r24
1c0:   ca 01        movw    r24, r20
1c2:   96 95        lsr     r25
1c4:   87 95        ror     r24
1c6:   96 95        lsr     r25
1c8:   87 95        ror     r24
1ca:   28 27        eor     r18, r24
1cc:   24 27        eor     r18, r20
1ce:   ca 01        movw    r24, r20
1d0:   96 95        lsr     r25
1d2:   87 95        ror     r24
1d4:   20 ff        sbrs    r18, 0
1d6:   03 c0        rjmp    .+6             ; 0x1de
1d8:   ac 01        movw    r20, r24
1da:   50 68        ori     r21, 0x80       ; 128
1dc:   01 c0        rjmp    .+2             ; 0x1e0
1de:   ac 01        movw    r20, r24
1e0:   50 93 17 01  sts     0x0117, r21
1e4:   40 93 16 01  sts     0x0116, r20
1e8:   6f bf        out     0x3f, r22       ; 63
```

Figure 16. Assembly Code for MaxmalLFSR

Appendix F: EBS Java Code

```
import java.io.*;
import java.util.Hashtable;
import java.util.Random;

public class EBS
{
  Random rndSim;
  Hashtable tblSourceToPubKey;
  Hashtable tblSourceToPrivKey;
  KeyStore[] arrKeyDict;
  SimLogger current;

  public static void main( String[] args )
  {
    EBS sim = new EBS();
    if ( args[0].equals( "-sim" ) )
    {
      long seed = Long.parseLong( args[1] );
      sim.runSim( seed );
    }
  }

  public EBS()
  {
    tblSourceToPubKey = new Hashtable();
    tblSourceToPrivKey = new Hashtable();
  }

  public void runSim( long seed )
  {
    rndSim = new Random( seed );

    // create the "motes" and load keys for each mote

    current = new SimLogger( 3, "Deterministic", 0 );
```

```
    simulateDeterministic( 3 );
    System.out.println( "Completed 3-deterministic experiment..." );
    current = new SimLogger( 6, "Deterministic", 0 );
    simulateDeterministic( 6 );
    System.out.println( "Completed 6-deterministic experiment..." );
    current = new SimLogger( 9, "Deterministic", 0 );
    simulateDeterministic( 9 );
    System.out.println( "Completed 9-deterministic experiment..." );
    current = new SimLogger( 27, "Deterministic", 0 );
    simulateDeterministic( 27 );
    System.out.println( "Completed 27-deterministic experiment..." );

    for ( int i=1; i < 11; i++ )
    {
      current = new SimLogger( 3, "Deterministic", i );
      simulateRandom( 3 );
      System.out.println( "Completed 3-random-" + i + " experiment..." );
    }
    for ( int i=1; i < 11; i++ )
    {
      current = new SimLogger( 6, "Deterministic", i );
      simulateRandom( 6 );
      System.out.println( "Completed 6-random-" + i + " experiment..." );
    }
    for ( int i=1; i < 11; i++ )
    {
      current = new SimLogger( 9, "Deterministic", i );
      simulateRandom( 9 );
      System.out.println( "Completed 9-random-" + i + " experiment..." );
    }
    for ( int i=1; i < 11; i++ )
    {
      current = new SimLogger( 27, "Deterministic", i );
      simulateRandom( 27 );
      System.out.println( "Completed 27-random-" + i + " experiment..." );
    }
}
```

```
public void simulateDeterministic( int size )
{
  current.simStart = System.currentTimeMillis();
  KeyStore ks = null;
  for ( int i=0; i < size; i++ )
  {
    String fname = i + ".ks";
    try
    {
      ks = new KeyStore( fname );
    }
    catch ( IOException ioex )
    {
      System.out.println( "Failed to load keystore." );
      return;
    }
    EBSPoint pubKey = ks.getKey( 0 );
    current.logKeySelectionf( i, ks.getLastIndex() );
    messageReceived( 65535, new KeyMessage( i, (short) 1, pubKey.getShortX() ) );
    messageReceived( 65535, new KeyMessage( i, (short) 0, pubKey.getShortY() ) );
  }
  current.simEnd = System.currentTimeMillis();
  current.writeResults();
}

public void simulateRandom( int size )
{
  current.simStart = System.currentTimeMillis();
  KeyStore ks = null;
  for ( int i=0; i < size; i++ )
  {
    String fname = i + ".ks";
    try
    {
      ks = new KeyStore( fname );
    }
    catch ( IOException ioex )
    {
```

```
      System.out.println( "Failed to load keystore." );
      return;
    }
    EBSPoint pubKey = ks.getRandomKey( rndSim );
    current.logKeySelectionf( i, ks.getLastIndex() );
    messageReceived( 65535, new KeyMessage( i, (short) 1, pubKey.getShortX() ) );
    messageReceived( 65535, new KeyMessage( i, (short) 0, pubKey.getShortY() ) );
  }
  current.simEnd = System.currentTimeMillis();
  current.writeResults();
}

public void messageReceived(int dstaddr, Message msg)
{
  if ( msg instanceof KeyMessage )
  {
    KeyMessage kmsg = (KeyMessage) msg;

    int sourceID = kmsg.get_sourceID();
    short isX = kmsg.get_isX();
    short[] coord = kmsg.get_coord();

    EBSPoint pubKey = null;
    Object tmp = tblSourceToPubKey.get( new Integer( sourceID ) );
    if ( tmp == null )
    {
      pubKey = new EBSPoint();
      tblSourceToPubKey.put( new Integer( sourceID ), pubKey );
    }
    else
    {
      pubKey = (EBSPoint) tmp;
    }
    if ( isX != 0 )
    {
      for ( int i=0; i < coord.length; i++ )
      {
        pubKey.x[i] = (byte) coord[i];
```

```
      }
    }
    else
    {
      for ( int i=0; i < coord.length; i++ )
      {
        pubKey.y[i] = (byte) coord[i];
      }
      //fire off encryption breaking thread
      findPrivateKey( sourceID );
    }
  }
}

private void findPrivateKey( int sourceID )
{
  long start = System.currentTimeMillis();

  boolean found = false;
  EBSPoint ActualKey = (EBSPoint) tblSourceToPubKey.get( new Integer( sourceID ) );
  EBSPoint G = EccM.initializePoint();

  int[] arrRndSequence = EccM.generateRandomSequence( sourceID );
  int index = 0;
  for ( ; index < arrRndSequence.length; index++ )
  {
    byte[] privKey = EccM.generatePrivateKey( index, arrRndSequence );

    EBSPoint trialKey = EccM.generatePublicKey( privKey );
    if ( ActualKey.isEqual( trialKey ) )
    {
      found = true;
      tblSourceToPrivKey.put( new Integer( sourceID ), privKey );
      break;
    }
  }
  long time = System.currentTimeMillis() - start;
  if ( !found )
```

```
      current.logResults( sourceID, ((long) -1), -1 );
    else
      current.logResults( sourceID, time, index+1 );
  }
}

/**
 * Implementation of ECC module.
 *
 * Heavily borrowed from Malan's EccM module in TinyOS
 */
import java.math.BigInteger;

public class EccM
{
  public static final String strOrder = "4000000000000000000020108a2e0cc0d99f8a5ef";
  public static final String strX = "2fe13c0537bbc11acaa07d793de4e6d5e5c94eee8";
  public static final String strY = "289070fb05d38ff58321f2e800536d538ccdaa3d9";

  public static final char[] digits = { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c',
'd', 'e', 'f' };
  public static final int NUMBITS = 163;
  public static final int NUMWORDS = (int) (2 * ((NUMBITS +1)/8.0 + 0.5));

  public static EBSCurve E = new EBSCurve();

  ////////////////////////////////////////////////////////////////////////
  // bint
  // routines
  ////////////////////////////////////////////////////////////////////////

  /**
   * Clears bint.
   */
  public static void b_clear(byte[] a)
  {
      a = new byte[NUMWORDS];
```

```
}

/**
 * Prints bint in hexadecimal to debugging console.
 */
public static void b_print(byte[] a)
{
  char[] output = new char[a.length*3];
  // iterate over bint's bytes, displaying each in hexadecimal
  for ( int i = 0; i < a.length; i++)
  {
    int b = a[i] & 0xff;
    output[i*3+1] = digits[b & 15];
    b = b >>> 4;
    output[i*3] = digits[b & 15];
    output[i*3+2] = ' ';
  }
  System.out.println( new String( output ) );
}


/**
 * Prints lower half of bint in hexadecimal to debugging console.
 */
public static void b_halfprint(byte[] a)
{
  char[] output = new char[a.length/2*3];
  // iterate over bint's bytes, displaying each in hexadecimal
  for ( int i = 0; i < a.length/2; i++)
  {
    int b = a[i+a.length/2] & 0xff;
    output[i*3+1] = digits[b & 15];
    b = b >>> 4;
    output[i*3] = digits[b & 15];
    output[i*3+2] = ' ';
  }
  System.out.println( new String( output ) );
}
```

```java
/**
 * Sets ith bit (where most significant bit is 0th bit) of bint.
 */
public static void b_setbit(byte[] a, int i)
{
  if ( a.length*8 < i )
    return; // not enough bits
  a[a.length - 1 - i/8] |= (1 << (i%8));
}


/**
 * Clears ith bit (where most significant bit is 0th bit) of bint.
 */
public static void b_clearbit(byte[] a, int i)
{
  if ( i == 0 )
    return;
  if ( a.length*8 < i )
    return; // not enough bits
  a[a.length - 1 - i/8] &= (0xff ^ (1 << (i % 8)));
}

/**
 * Returns TRUE iff bint is zero.
 */
public static boolean b_iszero(byte[] a)
{
  for (int i = 0; i < a.length; i++ )
    if ( a[i] != 0 )
      return false;

  return true;
}
```

```
/**
 * b = a.
 * Assumes a.length == b.length
 */
public static void b_copy(byte[] a, byte[] b)
{
  System.arraycopy( a, 0, b, 0, a.length );
}


/**
 * c = a XOR b.
 */
public static void b_xor(byte[] a, byte[] b, byte[] c)
{
  // let c[] = a[] XOR b[]; casting effectively unrolls loop a bit,
  // saving us some cycles
  for ( int i = 0; i < NUMWORDS; i++)
    c[i] = (byte) (a[i] ^ b[i]);
}


/**
 * Returns -1 if a < b, 0 if a == b, and 1 if a > b.
 */
public static int b_compareto(byte[] a, byte[] b)
{
  for ( int i = 0; i < NUMWORDS; i++ )
  {
    if ( a[i] != b[i] )
    {
      int x = a[i] & 0xff;
      int y = b[i] & 0xff;
      return x < y ? -1 : 1;
    }
  }
  return 0;
}
```

```
/**
 * Shifts bint left by n bits, storing result in b.
 *
 * a and b are allowed to point to the same memory.
 */
public static void b_shiftleft(byte[] a, int n, byte[] b)
{
  if ( n == 0 )
  {
    System.arraycopy( a, 0, b, 0, NUMWORDS );
    return;
  }

  // storage for shift's magnitudes
  int nBytes = n >>> 3;
  int nBits = n & 0x7;

  if ( nBytes != 0 )
  {
    for ( int i = nBytes; i < NUMWORDS; i++ )
      b[i-nBytes] = a[i];
    for ( int i = NUMWORDS - nBytes; i < NUMWORDS; i++ )
      b[i] = 0;
  }
  else if ( nBytes == 0 )
    System.arraycopy( a, 0, b, 0, NUMWORDS );

  int nBits2 = 8 - nBits;
  for ( int i = 1; i < NUMWORDS; i++ )
  {
    b[i-1] = (byte) (((b[i-1] << nBits) & 0xff) | ((b[i] & 0xff)>>> nBits2));
  }
  b[NUMWORDS-1] <<= nBits;
}


/**
```

```
 * Returns the number of bits in the shortest possible
 * representation of this bint.
 */
public static int b_bitlength( byte[] a )
{
  // local storage
  int n, x, y;

  // iterate over other bytes, looking for most significant set bit;
  // algorithm from Henry S. Warren Jr., Hacker's Delight
  for ( int i = 0; i < a.length; i++)
  {
    x = a[i] & 0xff;
    if ( x != 0)
    {
      n = 8;
      y = x >>> 4;
      if (y != 0)
      {
        n = n - 4; x = y;
      }
      y = x >>> 2;
      if (y != 0)
      {
        n = n - 2;
        x = y;
      }
      y = x >>> 1;
      if (y != 0)
        return (a.length - i - 1) * 8 + (8 - (n - 2));

      return (a.length - i - 1) * 8 + (8 - (n - x));
    }
  }

  // if no bits are set, bint is 0
  return 0;
}
```

```
/**
 * Returns TRUE iff ith bit of bint (where index of least
 * significant bit is 0) is set.  Recall that bints
 * are big-endian.
 */
public static boolean b_testbit(byte[] a, int i)
{
  return ( a[NUMWORDS - 1 - (i/8)] & ( 1 << (i % 8) ) ) != 0;
}


/**
 * Returns TRUE iff bints are equal.
 */
public static boolean b_isequal(byte[] a, byte[] b)
{
  // iterate over bints, looking for a difference
  for ( int i = 0; i < NUMWORDS; i++ )
    if (a[i] != b[i])
      return false;

  // if no difference found, bints are equal
  return true;
}

////////////////////////////////////////////////////////////////////////
// point
// routines
////////////////////////////////////////////////////////////////////////

/**
 * Clears point.
 */
public static void p_clear( EBSPoint P0 )
{
  // clear each coordinate
```

```
  b_clear( P0.x );
  b_clear( P0.y );
}

/**
 * Returns TRUE iff P0 == (0,0).
*/
public static boolean p_iszero( EBSPoint P0 )
{
  return b_iszero(P0.x) && b_iszero( P0.y );
}

/**
 * P1 = P0.
*/
public static void p_copy(EBSPoint P0, EBSPoint P1)
{
  // copy point's ordinates
  b_copy( P0.x, P1.x );
  b_copy( P0.y, P1.y );
}

/**
 * Prints point.
*/
public static void p_print( EBSPoint P0 )
{
  System.out.println( "x:" );
  b_halfprint( P0.x );
  System.out.println( "y:" );
  b_halfprint( P0.y );
}


//////////////////////////////////////////////////////////////////////////
// curve
// routines
//////////////////////////////////////////////////////////////////////////
```

```
/**
 * Multiplies P0 by n, storing result in P1.  P1 cannot be P0.
 *
 * Based on Algorithm IV.1 on p. 63 of "Elliptic Curves in Cryptography"
 * by I. F. Blake, G. Seroussi, N. P. Smart.
*/
public static void c_mul( byte[] n, EBSPoint P0, EBSPoint P1 )
{
  // index variable
  int i;

  // clear point
  p_clear( P1 );

  // perform multiplication
  for (i = b_bitlength(n) - 1; i >= 0; i--)
  {
    c_add( P1, P1, P1 );
    if ( b_testbit( n, i ) )
      c_add( P1, P0, P1 );
  }
}

/**
 * Q = P1 + P2.  Algorithm 7 in An Overview of Elliptic Curve Cryptography,
 * Lopez and Dahab.
 *
 * P1, P2, and Q are allowed to reference the same memory.
*/
public static void c_add( EBSPoint P1, EBSPoint P2, EBSPoint Q )
{
  byte[] lambda = new byte[NUMWORDS];
  byte[] numerator = new byte[NUMWORDS];
  EBSPoint T = new EBSPoint();

  // 1.  if P1 = 0
  if ( p_iszero( P1 ) )
```

```
{
  // Q <-- P2
  p_copy( P2, Q );
  return;
}

// 2.  if P2 = 0
if ( p_iszero( P2 ) )
{
  // Q <-- P1
  p_copy( P1, Q );
  return;
}

// 3.  if x1 = x2
if ( b_isequal( P1.x, P2.x ) )
{
  // if y1 = y2
  if ( b_isequal( P1.y, P2.y ) )
  {
    // lambda = x1 + y1/x1
    f_inv( P1.x, lambda );
    f_mul( lambda, P1.y, lambda );
    f_add( lambda, P1.x, lambda );

    // x3 = lambda^2 + lambda + a
    f_mul( lambda, lambda, T.x );
    f_add( T.x, lambda, T.x );
    f_add( T.x, E.a4, T.x );
  }
  else
  {
    // Q <-- 0
    b_clear( T.x );
    b_clear( T.y );
  }
}
else
```

```
  {
    // lambda <-- (y2 + y1)/(x2 + x1)
    f_add( P2.y, P1.y, numerator );
    f_add( P2.x, P1.x, lambda );
    f_inv( lambda, lambda );
    f_mul( numerator, lambda, lambda );

    // x3 <-- lambda^2 + lambda + x1 + x2 + a
    f_mul( lambda, lambda, T.x );
    f_add( T.x, lambda, T.x );
    f_add( T.x, P1.x, T.x );
    f_add( T.x, P2.x, T.x );
    f_add( T.x, E.a4, T.x );
  }

  // y3 <-- lambda(x1 + x2) + x3 + y1
  f_add( P1.x, T.x, T.y );
  f_mul( T.y, lambda, T.y );
  f_add( T.y, T.x, T.y );
  f_add( T.y, P1.y, T.y );

  // return
  p_copy( T, Q );
}


////////////////////////////////////////////////////////////////////////
// field
// routines
////////////////////////////////////////////////////////////////////////
/**
 * c = a + b.
 *
 * a, b, and/or c are allowed to point to the same memory.
 */
public static void f_add( byte[] a, byte[] b, byte[] c )
{
  b_xor( a, b, c );
```

```
}

/**
 * c = ab mod f
 *
 * Algorithm 4 from High-Speed Software Multiplication in F_{2^m}.
 *
 * a, b, and/or c are allowed to point to the same memory.
 */
public static void f_mul( byte[] a, byte[] b, byte[] c )
{
  // local variables
  byte[] T = new byte[NUMWORDS];

  // perform multiplication
  for ( int j = 7; j > -1; j-- )
  {
    for ( int i = 0; i <= NUMWORDS/2-1; i++ )
      if ( b_testbit( a, i*8+j ) )
        for ( int k = 0; k <= NUMWORDS/2-1; k++ )
          T[NUMWORDS - 1 - (k+i)] ^= b[NUMWORDS - 1 - k];
    if ( j != 0 )
      b_shiftleft(T, 1, T);
  }

  // modular reduction
  f_mod( T, c );
}

/**
 * b = a (mod modulus).
 *
 * a and b are allowed to point to the same memory.
 * Hardcoded at present with default curve's parameters to save cycles.
 */
public static void f_mod( byte[] a, byte[] b )
{
  // local variables
```

```
  int blr, shf;
  int comp;
  byte[] r = new byte[NUMWORDS];

  // modular reduction
  comp = b_compareto( a, E.modulus );
  if ( comp < 0 )
  {
    b_copy( a, b );
    return;
  }
  else if ( comp == 0 )
  {
    b_copy( r, b );
    return;
  }
  b_copy( a, r );
  blr = b_bitlength( r );
  while ( blr  >= E.bitlength )
  {
    shf = blr - E.bitlength;
    r[NUMWORDS - ((163+shf) / 8) - 1] ^= (1 << ((163+shf) % 8));
    r[NUMWORDS - (  (7+shf) / 8) - 1] ^= (1 << (  (7+shf) % 8));
    r[NUMWORDS - (  (6+shf) / 8) - 1] ^= (1 << (  (6+shf) % 8));
    r[NUMWORDS - (  (3+shf) / 8) - 1] ^= (1 << (  (3+shf) % 8));
    r[NUMWORDS - (  (0+shf) / 8) - 1] ^= (1 << (  (0+shf) % 8));
    blr = b_bitlength( r );
  }
  b_copy(r, b);
}

/**
 * d = a^-1.
 *
 * Algorithm 8 in "Software Implementation of Elliptic Curve Cryptography
 * Over Binary Fields", D. Hankerson, J.L. Hernandez, A. Menezes.
 *
 * a and d are allowed to point to the same memory.
```

```
*/
public static void f_inv(byte[] a, byte[] d)
{
  // local variables
  int i;
  int j;
  byte[] ptr;
  byte[][] anonymous = new byte[5][NUMWORDS];
  anonymous[0]     = new byte[NUMWORDS];
  byte[] b         = new byte[NUMWORDS];
  byte[] c         = new byte[NUMWORDS];
  byte[] u         = new byte[NUMWORDS];
  byte[] v         = new byte[NUMWORDS];
  anonymous[1] = b;
  anonymous[2] = c;
  anonymous[3] = u;
  anonymous[4] = v;

  // 1.  b <-- 1, c <-- 1, u <-- a, v <-- f
  for (i = 0; i < NUMWORDS; i++)
  {
    b[i] = 0;
    c[i] = 0;
    v[i] = E.modulus[i];
  }
  b[NUMWORDS-1] = 0x01;
  f_mod(a, u);

  // 2.  While deg(u) != 0
  int bitlen = b_bitlength(u);
  while ( bitlen > 1 )
  {
    // 2.1  j <-- deg(u) - deg(v).
    j = ( b_bitlength(u) - 1 ) - ( b_bitlength(v) - 1 );

    // 2.2  If j < 0 then:
    if ( j < 0 )
    {
```

```
  // u <--> v
  ptr = new byte[NUMWORDS];
  System.arraycopy( u, 0, ptr, 0, NUMWORDS );
  System.arraycopy( v, 0, u, 0, NUMWORDS );
  System.arraycopy( ptr, 0, v, 0, NUMWORDS );

  // b <--> c
  ptr = new byte[NUMWORDS];
  System.arraycopy( b, 0, ptr, 0, NUMWORDS );
  System.arraycopy( c, 0, b, 0, NUMWORDS );
  System.arraycopy( ptr, 0, c, 0, NUMWORDS );

  // j <-- -j
  j = -j;
}

// 2.3  u <-- u + x^jv
switch (j)
{
  case 0:
    f_add(u, v, u);
    f_add(b, c, b);
    break;
  case 1:
    b_shiftleft( v, 1, anonymous[0]);
    f_add(u, anonymous[0], u);
    b_shiftleft(c, 1, anonymous[0]);
    f_add(b, anonymous[0], b);
    break;
  case 2:
    b_shiftleft(v, 2, anonymous[0]);
    f_add(u, anonymous[0], u);
    b_shiftleft(c, 2, anonymous[0]);
    f_add(b, anonymous[0], b);
    break;
  default:
    b_shiftleft(v, j, anonymous[0]);
    f_add(u, anonymous[0], u);
```

```
        b_shiftleft(c, j, anonymous[0]);
        f_add(b, anonymous[0], b);
        break;
    }
    bitlen = b_bitlength(u);
  }
  b_copy(b, d);
}

/**
 * This function generates all possible public keys from the given sequence.
 * rndSequence is a pseudo-random sequence (PRS) generated from RandomLFSR
 * The function does not generate all possible permutations of rndSequence and assumes
 * the sequence is generated sequentially (i.e. atomically)
 */
public static EBSPoint[] generateKeys( int[] rndSequence )
{
  EBSPoint[] arrKeys = new EBSPoint[rndSequence.length];
  EBSPoint G = initializePoint();

  int len = rndSequence.length;
  for ( int index = 0; index < len; index++ )
  {
    byte[] privKey = new byte[NUMWORDS];
    for ( int i = NUMWORDS/2; i < NUMWORDS; i++ )
    {
      /**
       * The calculation for the index of rndSequence is as follows
       * index = the starting point of this random sequence
       * (i-NUMWORDS/2) = essentially becomes the increment value for running through rnd sequence
       * use modulo rndSequence.length to wrap around since index ranges from 0 thru
       * (rndSeqnence.length-1)
       */
      int eccNum = rndSequence[(index + ( i - NUMWORDS/2 )) % len];
      privKey[i] = (byte) eccNum;
    }

    /**
```

```
       * This section of code essentially sidesteps EccM's b_mod.
       * Initial tests indicate minimal performance degradation.  If performance becomes an issue, may
need
       * to revisit implementing b_mod.  Implementing b_mod in Java could prove difficult because of use
of
       * pointers when passing parameters to subfunctions.
       */
      BigInteger bint = new BigInteger( 1, privKey );
      BigInteger r = new BigInteger( strOrder, 16 );
      bint = bint.mod( r );
      byte[] z = bint.toByteArray();
      privKey = new byte[NUMWORDS];
      for ( int i = 0; i < z.length; i++ )
      {
        privKey[NUMWORDS - 1 - i] = z[z.length - 1 - i];
      }

      arrKeys[index] = new EBSPoint();
      c_mul( privKey, G, arrKeys[index] );
    }
    return arrKeys;
  }

  public static byte[] generatePrivateKey( int index, int[] rndSequence )
  {
    byte[] privKey = new byte[NUMWORDS];
    for ( int i = NUMWORDS/2; i < NUMWORDS; i++ )
    {
      int eccNum = rndSequence[(index + ( i - NUMWORDS/2 )) % rndSequence.length];
      privKey[i] = (byte) eccNum;
    }

    BigInteger bint = new BigInteger( 1, privKey );
    BigInteger r = new BigInteger( strOrder, 16 );
    bint = bint.mod( r );
    byte[] z = bint.toByteArray();
    privKey = new byte[NUMWORDS];
    for ( int i=0; i < z.length; i++ )
```

```
    privKey[NUMWORDS - 1 - i] = z[z.length - 1 - i];
  return privKey;
}

public static EBSPoint generatePublicKey( byte[] privKey )
{
  EBSPoint G = initializePoint();
  EBSPoint pubKey = new EBSPoint();

  c_mul( privKey, G, pubKey );
  return pubKey;
}

public static String printHex( BigInteger bi )
{
  String s = bi.toString( 16 );
  StringBuffer sb = new StringBuffer();
  int index = 0;
  if ( (s.length() % 2) == 1 )
  {
    index = 1;
    sb.append( "0" );
    sb.append( s.substring( 0, 1 ) );
    sb.append( " " );
  }
  while ( index < s.length() )
  {
    sb.append( s.substring( index, index+2 ) );
    sb.append( " " );
    index += 2;
  }
  return sb.substring( 0, sb.length()-1 );
}

public static String printHex( byte[] a )
{
  char[] output = new char[a.length*3];
  // iterate over bint's bytes, displaying each in hexadecimal
```

```
  for ( int i = 0; i < a.length; i++)
  {
    int b = a[i] & 0xff;
    output[i*3+1] = digits[b & 15];
    b = b >>> 4;
    output[i*3] = digits[b & 15];
    output[i*3+2] = ' ';
  }
  return new String( output );
}


public static EBSPoint initializePoint()
{
  EBSPoint G = new EBSPoint();

  // initilize Gx
  G.x[NUMWORDS - 21] = (byte) 0x02;
  G.x[NUMWORDS - 20] = (byte) 0xfe;
  G.x[NUMWORDS - 19] = (byte) 0x13;
  G.x[NUMWORDS - 18] = (byte) 0xc0;
  G.x[NUMWORDS - 17] = (byte) 0x53;
  G.x[NUMWORDS - 16] = (byte) 0x7b;
  G.x[NUMWORDS - 15] = (byte) 0xbc;
  G.x[NUMWORDS - 14] = (byte) 0x11;
  G.x[NUMWORDS - 13] = (byte) 0xac;
  G.x[NUMWORDS - 12] = (byte) 0xaa;
  G.x[NUMWORDS - 11] = (byte) 0x07;
  G.x[NUMWORDS - 10] = (byte) 0xd7;
  G.x[NUMWORDS - 9] = (byte) 0x93;
  G.x[NUMWORDS - 8] = (byte) 0xde;
  G.x[NUMWORDS - 7] = (byte) 0x4e;
  G.x[NUMWORDS - 6] = (byte) 0x6d;
  G.x[NUMWORDS - 5] = (byte) 0x5e;
  G.x[NUMWORDS - 4] = (byte) 0x5c;
  G.x[NUMWORDS - 3] = (byte) 0x94;
  G.x[NUMWORDS - 2] = (byte) 0xee;
  G.x[NUMWORDS - 1] = (byte) 0xe8;
```

```
  // initialize Gy
  G.y[NUMWORDS - 21] = (byte) 0x02;
  G.y[NUMWORDS - 20] = (byte) 0x89;
  G.y[NUMWORDS - 19] = (byte) 0x07;
  G.y[NUMWORDS - 18] = (byte) 0x0f;
  G.y[NUMWORDS - 17] = (byte) 0xb0;
  G.y[NUMWORDS - 16] = (byte) 0x5d;
  G.y[NUMWORDS - 15] = (byte) 0x38;
  G.y[NUMWORDS - 14] = (byte) 0xff;
  G.y[NUMWORDS - 13] = (byte) 0x58;
  G.y[NUMWORDS - 12] = (byte) 0x32;
  G.y[NUMWORDS - 11] = (byte) 0x1f;
  G.y[NUMWORDS - 10] = (byte) 0x2e;
  G.y[NUMWORDS - 9] = (byte) 0x80;
  G.y[NUMWORDS - 8] = (byte) 0x05;
  G.y[NUMWORDS - 7] = (byte) 0x36;
  G.y[NUMWORDS - 6] = (byte) 0xd5;
  G.y[NUMWORDS - 5] = (byte) 0x38;
  G.y[NUMWORDS - 4] = (byte) 0xcc;
  G.y[NUMWORDS - 3] = (byte) 0xda;
  G.y[NUMWORDS - 2] = (byte) 0xa3;
  G.y[NUMWORDS - 1] = (byte) 0xd9;

  return G;
}

public static String getPercent( int num, int den, int dec )
{
  double percent = (double) num;
  percent = percent/den * 100.0;
  String ret_val = String.valueOf( percent );
  int point = ret_val.indexOf( '.' );
  if ( point + dec > ret_val.length() )
    return ret_val.concat( getZeros( point+dec - ret_val.length() ) );
  return ret_val.substring( 0, point + dec );
}
```

```
  public static String getZeros( int len )
  {
    char[] c = new char[len];
    for ( int i = 0; i < len; i++ )
      c[i] = '0';
    return new String( c );
  }

  public static int[] generateRandomSequence( int seed )
  {
    RandomLFSR lfsr = new RandomLFSR( seed );

    int numUnique = 0;
    boolean[] rnd_vals = new boolean[65536];
    int[] temp = new int[65536];

    temp[numUnique] = lfsr.rand();
    while ( !rnd_vals[temp[numUnique]] )
    {
      rnd_vals[temp[numUnique]] = true; // indicate value has been generated
      temp[++numUnique] = lfsr.rand(); // get next "random" value
    }

    int[] arrRndSequence = new int[numUnique];
    System.arraycopy( temp, 0, arrRndSequence, 0, arrRndSequence.length );
    return arrRndSequence;
  }
}

import java.io.PrintWriter;
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class KeyStore
```

```
{
  int moteID;
  int[] arrRndSequence;
  EBSPoint[] arrPubKey;
  long generationTime;
  int sequenceLength;
  int lastIndex;

  public KeyStore( int mid )
  {
    moteID = mid;
    arrRndSequence = null;
    arrPubKey = null;

    generateKeys();
    sequenceLength = arrRndSequence.length;
    lastIndex = -1;
  }

  public KeyStore( String filename ) throws IOException
  {
    BufferedReader in = new BufferedReader( new FileReader( filename ) );
    String[] elements = in.readLine().split( " " );
    moteID = Integer.parseInt( elements[2] );

    elements = in.readLine().split( " " );
    generationTime = Long.parseLong( elements[2] );

    elements = in.readLine().split( " " );
    int len = Integer.parseInt( elements[3] );
    arrRndSequence = new int[len];
    arrPubKey = new EBSPoint[len];
    int i=0;
    while ( in.ready() )
    {
      elements = in.readLine().split( ";" );
      arrRndSequence[i] = Integer.parseInt( elements[1] );
      String[] arrX = elements[2].split( " " );
```

```
    String[] arrY = elements[3].split( " " );
    arrPubKey[i] = new EBSPoint();
    for ( int j=0; j < EccM.NUMWORDS; j++ )
    {
      arrPubKey[i].x[j] = (byte) Integer.valueOf( arrX[j], 16 ).intValue();
      arrPubKey[i].y[j] = (byte) Integer.valueOf( arrY[j], 16 ).intValue();
    }
    i++;
  }
  sequenceLength = arrRndSequence.length;
}

private void generateKeys()
{
  long time = System.currentTimeMillis();

  // create the random sequence
  arrRndSequence = EccM.generateRandomSequence( moteID );
  arrPubKey = EccM.generateKeys( arrRndSequence );

  generationTime = System.currentTimeMillis() - time;
}

public EBSPoint getKey( int index )
{
  lastIndex = index;
  return arrPubKey[index];
}

public EBSPoint getRandomKey( Random prng )
{
  lastIndex = prng.nextInt( sequenceLength );
  return arrPubKey[index];
}

public int getLastIndex()
{
  return lastIndex;
```

```
  }

  public void writeKeys( String filename ) throws IOException
  {
    PrintWriter out = new PrintWriter(new BufferedWriter(new FileWriter(filename)));
    out.println( "Mote ID: " + moteID );
    out.println( "Generation Time: " + generationTime + " msecs" );
    out.println( "Length of Sequence: " + sequenceLength );
    for ( int i=0; i < sequenceLength; i++ )
    {
      out.println( i + ";" + arrRndSequence[i] + ";" + EccM.printHex( arrPubKey[i].x ) + ";" +
EccM.printHex( arrPubKey[i].y ) );
    }
    out.close();
  }

}

public class RandomLFSR
{
  private int TOS_LOCAL_ADDRESS;
  private int shiftReg;
  private int initSeed;
  private int mask;

  public RandomLFSR( int seed )
  {
    TOS_LOCAL_ADDRESS = seed;
    /* Initialize the seed from the ID of the node */
    //System.out.println( "RANDOM_LFSR initialized." );
    shiftReg = 119 * 119 * (TOS_LOCAL_ADDRESS + 1);
    shiftReg = shiftReg & 0xffff;
    initSeed = shiftReg;
    mask = 137 * 29 * (TOS_LOCAL_ADDRESS + 1);
    mask = mask & 0xffff;

    //for ( int i=0; i<65536; i++ )
    //{
```

```
    //  System.out.println( rand() );
    //}
  }
  public int rand()
  {
    /* Return the next 16 bit random number */
    boolean endbit;
    int tmpShiftReg;
    tmpShiftReg = shiftReg;
    endbit = ((tmpShiftReg & 0x8000) != 0);
    tmpShiftReg = tmpShiftReg << 1;
    if ( endbit )
      tmpShiftReg = tmpShiftReg ^ 0x100b;
    tmpShiftReg++;
    tmpShiftReg = tmpShiftReg & 0xffff;
    shiftReg = tmpShiftReg;
    tmpShiftReg = tmpShiftReg ^ mask;
    return tmpShiftReg;
  }
}

public class EBSPoint
{
  public byte[] x;
  public byte[] y;

  public EBSPoint()
  {
    x = new byte[EccM.NUMWORDS];
    y = new byte[EccM.NUMWORDS];
  }

  public EBSPoint p_copy()
  {
    EBSPoint copy = new EBSPoint();
    System.arraycopy( x, 0, copy.x, 0, EccM.NUMWORDS );
    System.arraycopy( y, 0, copy.y, 0, EccM.NUMWORDS );
    return copy;
```

```
  }

  public boolean isZero()
  {
    for (int i = 0; i < EccM.NUMWORDS; i++ )
      if ( x[i] != 0 || y[i] != 0)
        return false;

    return true;
  }

  public boolean isEqual( EBSPoint ebsp )
  {
    return EccM.b_isequal( x, ebsp.x ) && EccM.b_isequal( y, ebsp.y );
  }

  public short[] getShortX()
  {
    short[] sx = new short[EccM.NUMWORDS];
    for ( int i=0; i < x.length; i++ )
    {
      sx[i] = (short) (x[i] & 0xff);
    }
    return sx;
  }

  public short[] getShortY()
  {
    short[] sy = new short[EccM.NUMWORDS];
    for ( int i=0; i < y.length; i++ )
    {
      sy[i] = (short) (y[i] & 0xff);
    }
    return sy;
  }
}
```

```
import java.math.BigInteger;

public class EBSCurve
{
  public byte[] a4;
  public byte[] a6;

  public byte[] modulus;

  public int bitlength;

  public EBSCurve()
  {
    a4 = new byte[EccM.NUMWORDS];
    a4[EccM.NUMWORDS-1] = 1;
    a6 = new byte[EccM.NUMWORDS];
    a6[EccM.NUMWORDS-1] = 1;
    modulus = new byte[EccM.NUMWORDS];

    EccM.b_setbit( modulus, 163 );
    EccM.b_setbit( modulus, 7 );
    EccM.b_setbit( modulus, 6 );
    EccM.b_setbit( modulus, 3 );
    EccM.b_setbit( modulus, 0 );
    bitlength = 164;
  }
}

public class Message { }

public class KeyMessage extends Message
{
  int sourceID;
  short isX;
  short[] coord;

  public KeyMessage( int id, short x, short[] c )
```

```
  {
    sourceID = id;
    isX = x;
    coord = c;
  }

  public int get_sourceID()
  {
    return sourceID;
  }

  public short get_isX()
  {
    return isX;
  }

  public short[] get_coord()
  {
    return coord;
  }
}

import java.io.*;

public class SimLogger
{
  int size;
  String simType;
  int experiment;
  int[] keySelection;
  long[] times;
  int[] trials;

  long simStart;
  long simEnd;

  public SimLogger( int size, String simType, int experiment )
```

```
{
  this.size = size;
  this.simType = simType;
  this.experiment = experiment;
  keySelection[size];
  times = new long[size];
  trials = new int[size];
}

public void logResults( int mote, long time, int tries )
{
  times[mote] = time;
  trials[mote] = tries;
}

public void logKeySelection( int mote, int selection )
{
  keySelection[mote] = selection;
}

public void writeResults()
{
  try
  {
    String filename = size + "-" + simType + "-" + experiment + ".sr";
    PrintWriter out = new PrintWriter(new BufferedWriter(new FileWriter(filename)));
    out.println( "Network size: " + size );
    out.println( "Sim Time: " + (simEnd - simStart) );
    for ( int i=0; i < size; i++ )
    {
      out.println( i + ";" + times[i] + ";" + trials[i] + ";" + keySelection[i] );
    }
    out.close();
  }
  catch ( IOException ioex )
  {
    ioex.printStackTrace();
  }
```

```
  }
}
```

Appendix F: MaximalLFSR Code

```
module MaximalLFSR
{
  provides interface Random;
}
implementation
{
  uint16_t shiftReg;
  uint16_t initSeed;

  /* Initialize the seed from the ID of the node */
  async command result_t Random.init() {
    dbg(DBG_BOOT, "RANDOM_LFSR initialized.\n");
    atomic {
      shiftReg = (TOS_LOCAL_ADDRESS + 1);
      initSeed = shiftReg;
    }
    return SUCCESS;
  }

  async command result_t Random.initseed(uint16_t s) {
    atomic {
      shiftReg = (s + 1);
      initSeed = shiftReg;
    }
    return SUCCESS;
  }

  /* Return the next 16 bit random number */
  async command uint16_t Random.rand() {
    bool endbit;
    uint16_t tmpShiftReg;

    atomic {
      tmpShiftReg = shiftReg;
```

```
            endbit = ((tmpShiftReg >> 16) ^ (tmpShiftReg >> 5) ^ (tmpShiftReg >> 3) ^
                      (tmpShiftReg >> 2)  ^ (tmpShiftReg)) & 0x0001;
      if (endbit)
            tmpShiftReg = (tmpShiftReg >> 1) | 0x8000;
      else
        tmpShiftReg = tmpShiftReg >> 1;
      shiftReg = tmpShiftReg;
    }
    return tmpShiftReg;
  }

  async command uint32_t Random.rand32() {
    return (uint32_t)call Random.rand() << 16 | call Random.rand();
  }

  uint16_t TOSH_rand() __attribute__((C)) {
    return call Random.rand();
  }
}
```

Bibliography

[ASS01] Akyildiz, I.F., Su, W., Sankarasubramaniam, Y., and Cayirci, E. "Wireless Sensor Networks: A Survey," *Computer Networks: The International Journal of Computer and Telecommunications Networking*. Vol. 38, No. 4: 393-422 (March 2002).

[Atm04] *Atmel*. Atmega128 Datasheet, revision M. (November 2004). 31 Jan 2006 http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf

[BBB05] Barker, E., Barker, W., Burr, W., Polk, W., and Smid, M. *Special Publication 800-57: Recommendation for Key Management.* National Institute of Standards and Technology, August 2005.

[Bis03] Bishop, Matt. *Computer Security: Art and Science* (5th Edition). Boston: Pearson Education, Inc, 2003.

[Blo85] Blom, R. "An Optimal Class of Symmetric Key Generation Systems," *Advances in Cryptology: Proceedings of EUROCRYPT 84, Lecture Notes in Computer Science.* 335-338. New York: Springer-Verlag, 1985.

[BSH98] Blundo, C., De Santis, A., Herzberg, A., Kutten, S., Vaccaro, U., and Yung, M. "Perfectly Secure Key Distribution for Dynamic Conferences," *Information and Computation.* Vol 146, No 1: 1-23 (October 1998).

[CKM00] Carman, D., Kruus, P., and Matt, B. *Constraints and Approaches for Sensor Network Security*. NAI Labs Technical Report 00-010. Glenwood MD: Network Associates, Inc., September, 2000.

[CoM67] Coveyou, R.R. and MacPherson, R.D. "Fourier Analysis of Uniform Random Number Generators," *Journal of the ACM*. Vol. 14, No. 1: 100-119 (January 1967).

[CPS03] Chan, H., Perrig, A., and Song, D. "Random Key Predistribution Schemes for Sensor Networks," *IEEE Symposium on Security and Privacy*. 197. Washington, D.C.: IEEE Computer Society, 2003.

[DDH03] Du, W., Deng, J., Han, Y., and Varshney, P. "A Pairwise Key Pre-Distribution Scheme for Wireless Sensor Networks," *Proceedings of the Tenth ACM Conference on Computer and Communications Security.* 42-51. New York: ACM Press, 2003.

[DCL04] Duetre, B., Cheung, S., and Levy, J. *Lightweight Key Management in Wireless Sensor Networks by Leveraging Initial Trust.* Contract F30602-02-C-0212. Menlo Park CA: SRI International, April 2004.

[DiH76] Diffie, W. and Hellman, M. E. "New Directions in Cryptography," *IEEE Transactions on Information Theory*, Vol. IT-22, No. 6: 644-654 (November 1976).

[EsG02] Eschenauer, L. and Gligor, V. D. "A Key-Management Scheme for Distributed Sensor Networks," *Proceedings of the Ninth ACM Conference on Computer and Communication Security*. 41-47. New York: ACM Press, 2002.

[FiM86] Fishman, G.S. and Moore, L.R. "An Exhaustive Analysis of Multiplicative Congruential Random Number Generators with Modulus 2^{31}-1," *SIAM Journal on Scientific and Statistical Computing*. Vol. 7, No. 1: 24-25 (January 1986).

[GLB03] Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., and Culler, D. "The *nesC* Language: A Holistic Approach to Network Embedded Systems," *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. 1-11. New York: ACM Press, 2003.

[HiC02] Hill, J.L. and Culler, D.E. "Mica: A Wireless Platform for Deeply Embedded Networks," *IEEE Micro*, Vol. 22, No. 6: 12-24 (November 2002).

[HHK04] Hill, J., Horton, M., Kling, R., and Krishnamurthy, L. "The Platforms Enabling Wireless Sensor Networks," *Communications of the ACM*, Vol. 47, No. 6: 41-46 (June 2004).

[Hoa76] Hoaglin, D.C. Theoretical Properties of Congruential Random-Number Generators: An Empirical View. Memorandum NS-340. Department of Statistics, Harvard University. Cambridge, MA, 1976.

[HSW00] Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., and Pister, K. "System Architecture Directions for Networked Sensors," *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*. 93-104. New York: ACM Press, 2000.

[HPJ03] Hu, Y., Perrig, A., and Johnson, D.B. "Rushing Attacks and Defense in Wireless Ad Hoc Network Routing Protocols," *Proceedings of the 2003 ACM Workshop on Wireless Security*. 30-40. New York: ACM Press, 2003.

[HLV04] Hwang, D. D., Lai, B. C., and Verbauwhede, I. "Energy-Memory-Security Tradeoffs in Distributed Sensor Networks," *Proceedings of the Third International Conference on Ad-Hoc, Mobile, and Wireless Networks (ADHOC-NOW)*. 70-81.

[Joh94] Johnson, D.B. "Routing in Ad Hoc Networks of Mobile Hosts," *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications*. 158-163. IEEE Computer Society, December 1994.

[KaY98] Kaliski, B. and Yin, Y. *On the Security of the RC5 Encryption Algorithm*. RSA Laboratories Technical Report TR-602. Bedford, MA: RSA Laboratories, September 1998.

[KaW03] Karlof, C. and Wagner, D. "Secure routing in wireless sensor networks: Attacks and countermeasures," *Proceedings of the First IEEE International Workshop on Sensor Network Protocols and Applications*. 113-127. May 2003.

[KKP99] Kahn, J. M., Katz, R. H., and Pister, K. S. J. "Next century challenges: mobile networking for 'Smart Dust'," *Proceedings of the Fifth ACM/IEEE International Conference on Mobile Computing and Networking*. 271-278. New York: ACM Press, 1999.

[Knu81] Knuth, D.E. *The Art of Computer Programming* (2nd Edition). Reading, MA: Addison-Wesley, 1981.

[KSW04] Karlof, C., Sastry, N., and Wagner, D. "TinySec: A Link Layer Security Architecture for Wireless Sensor Networks," *Proceedings of the Second Conference on Embedded Networked Systems*. 162-175. New York: ACM Press, 2004.

[Lev04] Levis, P. "RandomMLCG," Electronic Message. 22:08:11 PDT, 2 Sep 2004. 31 Jan 2006 https://mail.millennium.berkeley.edu/pipermail/tinyos-devel/2004-September/000500.html

[LiN03] Liu, D. and Ning, P. "Establishing Pairwise Keys in Distributed Sensor Networks," *Proceedings of the Tenth ACM Conference on Computer and Communications Security*. 52-61. New York: ACM Press, 2003.

[Mal04] Malan, D. *Crypto for Tiny Objects*. Harvard University Technical Report TR-04-04. Cambridge, MA: Harvard University, January 2004.

[Mil86] Miller, V. "Use of Elliptic Curves in Cryptography," *Advances in Cryptology: Proceedings of CRYPTO 85, Lecture Notes in Computer Science*. 417-426. New York: Springer-Verlag, 1986.

[MPS02] Mainwaring, A., Polastre, J., Szewczyk, R., Culler, D., and Anderson, J. "Wireless Sensor Networks for Habitat Monitoring," 2002

[MWS04] Malan, D. J., Welsh, M., and Smith, M. D. "A Public key Infrastructure for Key Distribution in TinyOS Based on Elliptic Curve Cryptography," *In the First Conference on Sensor and Ad Hoc Communications and Networks*. 2004

[NBS99] National Bureau of Standards. *Data Encryption Standard*. FIPS-Pub.46-3. Washington D.C.: U.S. Department of Commerce, October 1999.

[Nel99] Nelson, M. "56-bit DES Algorithm Broken in Record Time," *InfoWorld*, Vol. 21, No. 4: 45 (January 1999).

[NSS04] Newsome, J., Shi, E., Song, D., and Perrig, A. "The Sybil Attack in Sensor Networks: Analysis and Defenses," *Proceedings of the Third International Symposium on Information Processing in Sensor Networks*. 259-268. New York: ACM Press, 2004.

[PaM88] Park, S. K. and Miller, K. W. "Random Number Generators: Good Ones are Hard to Find," *Communications of the ACM*, Vol. 31, No. 10: 1192-1201 (October 1988).

[PeR99] Perkins, C.E. and Royer, E.M. "Ad Hoc On-Demand Distance Vector Routing," *Proceedings of the Second IEEE Workshop on Mobile Computing Systems and Applications*. 90-100. February 1999.

[PSW01] Perrig, A., Szewczyck, R., Wen, V., Culler, D., and Tygar, J.D. "SPINS: Security Protocols for Sensor Networks," *Proceedings of the Seventh Annual International Conference on Mobile Computing*. 189-199. New York: ACM Press, 2001.

[PSW04] Perrig, A., Stankovic, J. and Wagner, D. "Security in Wireless Sesnor Networks," *Communications of the ACM*, Vol. 47, No. 6: 53-57 (June 2004).

[PTS00] Perrig, A., Canetti, R., Tygar, J. D., and Song, D. Efficient Authentication and Signing of Multicast Streams Over Lossy Channels. *Proceedings of the 2000 IEEE Symposium on Security and Privacy*. 56. Washington, D.C.: IEEE Computer Society, 2000.

[PoK00] Pottie, G.J. and Kaiser, W.J. "Wireless Integrated Network Sensors," *Communications of the ACM*, Vol. 43, No. 5: 51-58 (June 2004).

[RSA78] Rivest, R.L., Shamir, A., and Adleman, L. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Communications of the ACM*. Vol. 21, No. 2: 120-126 (February 1978).

[Sch96] Schneier, B. *Applied Cryptography* (2nd Edition). New York: John Wiley & Sons, Inc, 1996.

[SHC04] Shnayder, V., Hempstead, M., Chen, B., Allen, G. W., and Welsh, M. "Simulating the Power Consumption of Large-Scale Sensor Network Applications," *Proceedings of the Second International Conference on Embedded Networked Sensor Systems*. 188-200. New York: ACM Press, 2004.

[SOP04] Szewczyk, R., Osterweil, E., Polastre, J., Hamilton, M., Mainwaring, A., and Estrin, D. "Habitat Monitoring with Sensor Networks," *Communications of the ACM*, Vol. 47, No. 6: 34-40 (June 2004).

[SPM04] Szewczyk, R., Polastre, J., Mainwaring, A., and Culler, D. "Lessons from a Sensor Network Expedition," 2004.

[Sti02] Stinson, D.R. *Cryptography: Theory and Practice*. Boca Raton: Chapman & Hall/CRC, 2002.

[TAH02] Tilak, S., Abu-Ghazaleh, N. B., and Heinzelman, W. "A Taxonomy of Wireless Micro-sensor Network Models," *ACM SIGMOBILE Mobile Computing and Communications Review*, Vol. 6, No. 2: 28-36 (April 2002).

[Tin05] TinyOS Community Forum. "TinyOS 2.0 PreRelease2," (November 2005) 12 February 2006 http://www.tinyos.net/scoop/story/2005/11/1/11215/8473

[Wag03] Wagstaff, S.S., Jr. *Cryptanalysis of Number Theoretic Ciphers*. Boca Raton: Chapman & Hall/CRC, 2003.

[WKC04] Watro, R., Kong, D., Cuti, S., Gardiner, C., Lynn, C., and Kruus, P. "TinyPK: Securing Sensor Networks with Public Key Technology," *Proceedings of the Second ACM Workshop on Security of Ad Hoc and Sensor Networks*. 59-64. New York: ACM Press, 2004.

[WoS02] Wood, A. and Stankovic, J. "Denial of Service Attacks in Sensor Networks," *Computer*, Vol. 35, No. 10: 54-62 (October 2002).

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From – To)*
23 March 2006	Master's Thesis	March 2005 – March 2006

4. TITLE AND SUBTITLE

Cryptanalysis of Pseudorandom Number Generators in Wireless Sensor Networks

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Finnigin, Kevin M., 1st Lt, USAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way, Building 640
WPAFB OH 45433-8865

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GIA/ENG/06-05

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Security Agency
9800 Savage Rd., Suite 9704
Ft. Meade, MD 20755-6704
POC: Neal Ziring
(410) 854-5762

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This work presents a brute-force attack on an elliptic curve cryptosystem implemented on UC Berkley's TinyOS operating system for wireless sensor networks. The attack exploits the short period of the pseudorandom number generator (PRNG) used by the cryptosystem to generate private keys. The attack assumes a laptop is listening promiscuously to network traffic for key messages and requires only the sensor node's public key and network address to discover the private key. Experimental results show that roughly 50% of the address space leads to a private key compromise in 25 minutes on average. Furthermore, approximately 32% of the address space leads to a compromise in 17 minutes on average, 11% in 6 minutes, and the remaining 7% in 2 minutes or less. Two alternatives to the PRNG are examined that mitigate the brute-force attack. The alternatives are implemented on the Mica2 mote and examined to determine CPU cycles for execution and memory requirements. The recommended PRNG requires 73 CPU cycles in the worst case and uses 66 bytes of memory. The period of the PRNG is uniform for all mote addresses and theoretically requires 6.6 years on average for a key compromise for the attack used in this thesis.

15. SUBJECT TERMS

Computer Networks, Cryptography, Data Transmission Security, Random Number Generators, Wireless Sensor Networks

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Barry E. Mullins, Ph.D, AFIT/ENG
U	**U**	**U**	**UU**	135	**19b. TELEPHONE NUMBER** *(Include area code)* 785-3636, ext 7979 (barry.mullins@afit.edu)

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18